AndeStar™ V5 Instruction Extension Specification

Document Number UM165-155

Date Issued 2025-03-17

Copyright Notice

Contact Information

Should you have any problems with the information contained herein, you may contact Andes Technology Corporation through

- email – support@andestech.com
- Website – https://es.andestech.com/eservice/

Please include the following information in your inquiries:

- the document title
- the document number

- the page number(s) to which your comments apply
- a concise explanation of the problem

General suggestions for improvements are welcome.

Revision History

Rev.	Revision Date	Revised Content
1.5.8	2025/03/17	1. Use the new version (2025) of Andes logo.
1.5.7	2024/12/11	2. Separate XAndesCoDense to XAndesCoDense(NDS.EXEC.IT) and XAndesNewCoDense(NDS.NEXEC.IT).
1.5.6	2024/09/30	3. Add the missing word "{" in the pseudo code of BFOS and BFOZ. 4. Explain why there are two versions (Andes and RISC-V) of the two instructions (FCVT.S.BF16 and FCVT.BF16.S).
1.5.5	2024/05/15	5. Removed ex9.it from exec.it instruction. (Section 3.2.1)
1.5.4	2024/05/13	1. Fixed the conflict description between the operation 2 and pseudo code for NDS.EXEC.IT instruction. (Section 3.2.1)
1.5.3	2024/02/16	1. Added the prefix “NDS” for each instruction. (All sections) 2. Gave an abbreviated name for each extension. (Chapter 1)
1.5.2	2023/10/23	1. Allowed VLN8.V, VLNU8.V, VLE4.V to process elements from the index of VSTART and generate new exceptions for load page fault. (Section 3.6.1, 3.6.2, 3.9.1)
1.5	2023/06/29	1. Removed intrinsic functions for vector bfloat16 conversion. (Section 3.5.1, 3.5.2) 2. Updated the table for the combination of vd LMUL and vs EMUL. (Section 3.9.2, 3.9.3, 3.9.4, 3.9.5) 3. Fixed the condition of 4-bit EEW support for VZEXT and VSEXT. (Section 3.9) 4. Added Vector Dot Product extension. (Section 2.8, 3.8)

Rev.	Revision Date	Revised Content
		5. Added Andes Vector Small INT Handling extension (Section 2.9, 3.9) 6. Added Andes Vector Quad-Widening Integer Multiply-Add extension (Section 2.10, 3.10) 7. Fixed the behavior of FCVT.S.BF16 for NaN problem (Section 3.4.1) 8. Added new nexec.it instruction, which is exactly the same as exec.it but with different opcode encoding (Section 1.1). 9. Added vector packed FP16 extension instructions. (Section 2.7, 3.7)
1.4	2020/11/19	1. Used __bf16 for the type of bfloat16 data. (Section 3.4.1, 3.4.2) 2. Added intrinsic functions for vector bfloat16 conversion. (Section 3.5.1, 3.5.2)
1.3	2020/10/12	1. Added INT4 Vector Load extension. (Section 2.6 and 3.6) 2. Added scalar and vector BFLOAT16 conversion extensions. (Section 2.4, 2.5, 3.4, 3.5)
1.2	2019/7/18	Removed the support support for FLHW/FSHW instructions and moved their descriptions to the Appendix section as they conflict with Vector extension encoding.
1.1	2018/6/20	1. Added FLHW and FSHW instructions. 2. Extended ex9.it to exec.it instruction. (Section 3.2.1)
1.0	2017/11/17	Initial Release

Table of Contents

List of Tables

Typographical Convention Index

Document Element	Font	Font Style	Size	Color
Normal text	Georgia	Normal	12	Black
`Command line, source code or file paths`	`Lucida Console`	`Normal`	11	`Indigo`
`VARIABLES OR PARAMETERS IN COMMAND LINE, SOURCE CODE OR FILE PATHS`	**`LUCIDA CONSOLE`**	**`BOLD + ALL-CAPS`**	11	**`INDIGO`**
Hyperlink	Georgia	Underlined	12	Blue

1. Introduction

To meet users' wide-ranged requirements and help accelerate the adoption of RISC-V architecture in the embedded markets, the AndeStar V5 ISA architecture extends the RISC-V base ISA architecture with Andes instruction extensions. It includes the following components:

- RISC-V base ISA and standard extensions
  - RISC-V RVI base integer instruction set
  - RISC-V RVC standard extension for compressed instructions
  - RISC-V RVM standard extension for integer multiplication and division
  - Optional RISC-V RVA standard extension for atomic operations
- Andes-extended ISA extensions
  - Andes Performance extension (XAndesPerf)
  - Andes CoDense extension (XAndesCoDense)
  - Andes New CoDense extension (XAndesNewCoDense)
  - Andes Scalar BFLOAT16 Conversion extension (XAndesBFHCvt)
  - Andes Vector BFLOAT16 Conversion extension (XAndesVBFHCvt)
  - Andes Vector INT4 Load extension (XAndesVSIntLoad)
  - Andes Vector Packed FP16 extension (XAndesVPackFPH)
  - Andes Vector Dot Product extension (XAndesVDot)
  - Andes Vector Small INT Handling extension (XAndesVSIntH)
  - Andes Vector Quad-Widening Integer Multiply-Add extension (XAndesVQMac)

This document introduces the instructions in the Andes-extended ISA extensions. These instructions are designed for compilers or library developers to enhance application performance and reduce code size. For details of instructions in the RISC-V base ISA and standard extensions, see the *RISC-V Instruction Set Manual*, including *Volume I: User-Level ISA* and *Volume II: Privileged Architecture*, on the RISC-V website.

2. Andes Instruction Summary

2.1. Andes Performance Extension (XAndesPerf)

Table 1. Branch Instructions

No.	Arch.	Mnemonic	Instruction	Operation
1	RV32	NDS.BBC *Rs1, #cimm[4:0], #imm[10:1]*	Branch on test bit is clear/zero	`if (Rs1[cimm] == 0) { tPC = PC + SE(CONCAT(imm[10:1],0[0])); PC = tPC }`
	RV64	NDS.BBC *Rs1, #cimm[5:0], #imm[10:1]*		
2	RV32	NDS.BBS *Rs1, #cimm[4:0], #imm[10:1]*	Branch on test bit is set/not zero	`if (Rs1[cimm] != 0) { tPC = PC + SE(CONCAT(imm[10:1],0[0])); PC = tPC }`
	RV64	NDS.BBS *Rs1, #cimm[5:0], #imm[10:1]*		
3	RV32 & RV64	NDS.BEQC *Rs1, #cimm[6:0], #imm[10:1]*	Branch on equal to constant	`if (Rs1 == ZE(cimm)) { tPC = PC + SE(CONCAT(imm[10:1],0[0])); PC = tPC }`
4	RV32 & RV64	NDS.BNEC *Rs1, #cimm[6:0], #imm[10:1]*	Branch on not equal to constant	`if (Rs1 != ZE(cimm)) { tPC = PC + SE(CONCAT(imm[10:1],0[0])); PC = tPC }`

				}
5	RV32	NDS.BFOS *Rd, Rs1, #msb[4:0], #lsb[4:0]*	Sign-extended bit field operation	See page 25
	RV64	NDS.BFOS *Rd, Rs1, #msb[5:0], #lsb[5:0]*		
6	RV32	NDS.BFOZ *Rd, Rs1, #msb[4:0], #lsb[4:0]*	Zero-extended bit field operation	See page 29
	RV64	NDS.BFOZ *Rd, Rs1, #msb[5:0], #lsb[5:0]*		

Table 2. Load Effective Address Instructions

No.	Arch.	Mnemonic	Instruction	Operation
1	RV32 & RV64	NDS.LEA.H *Rd, Rs1, Rs2*	Load effective half-word address	Rd = Rs1 + Rs2*2
2	RV32 & RV64	NDS.LEA.W *Rd, Rs1, Rs2*	Load effective word address	Rd = Rs1 + Rs2*4
3	RV32 & RV64	NDS.LEA.D *Rd, Rs1, Rs2*	Load effective double-word address	Rd = Rs1 + Rs2*8
4	RV64	NDS.LEA.B.ZE *Rd, Rs1, Rs2*	Load effective byte address from unsigned 32-bit	Rd = Rs1 + ZE32(Rs2[31:0])

			offset	
5	RV64	`NDS.LEA.H.ZE` *Rd, Rs1, Rs2*	Load effective half-word address from unsigned 32-bit offset	`Rd = Rs1 + ZE32(Rs2[31:0])*2`
6	RV64	`NDS.LEA.W.ZE` *Rd, Rs1, Rs2*	Load effective word address from unsigned 32-bit offset	`Rd = Rs1 + ZE32(Rs2[31:0])*4`
7	RV64	`NDS.LEA.D.ZE` *Rd, Rs1, Rs2*	Load effective double-word address from unsigned 32-bit offset	`Rd = Rs1 + ZE32(Rs2[31:0])*8`

Table 3. Global Pointer(GP)–Relative Instructions

No.	Arch.	Mnemonic	Instruction	Operation
1	RV32 & RV64	`NDS.ADDIGP Rd, imm[17:0]`	GP-implied add immediate	`Rd = x3 + SE(imm[17:0])`
2	RV32 & RV64	`NDS.LBGP Rd, [+ imm[17:0]]`	GP-implied load byte signed immediate	See page 40
3	RV32 & RV64	`NDS.LBUGP Rd, [+ imm[17:0]]`	GP-implied load byte unsigned immediate	See page 43
4	RV32 & RV64	`NDS.LHGP Rd, [+ (imm[17:1] << 1)]`	GP-implied load half-word signed immediate	See page 45
5	RV32 & RV64	`NDS.LHUGP Rd, [+ (imm[17:1] << 1)]`	GP-implied load half-word unsigned immediate	See page 47
6	RV32 & RV64	`NDS.LWGP Rd, [+ (imm[18:2] << 2)]`	GP-implied load word signed immediate	See page 49
7	RV64	`NDS.LWUGP Rd, [+ (imm[18:2] << 2)]`	GP-implied load word unsigned immediate	See page 51
8	RV64	`NDS.LDGP Rd, [+ (imm[19:3] << 3)]`	GP-implied load double-word immediate	See page 53
9	RV32 & RV64	`NDS.SBGP Rs2, [+ imm[17:0]]`	GP-implied store byte immediate	See page 55

No.	Arch.	Mnemonic	Instruction	Operation
10	RV32 & RV64	NDS.SHGP *Rs2, [+ (imm[17:1] << 1)]*	GP-implied store half-word immediate	See page 57
11	RV32 & RV64	NDS.SWGP *Rs2, [+ (imm[18:2] << 2)]*	GP-implied store word immediate	See page 59
12	RV64	NDS.SDGP *Rs2, [+ (imm[19:3] << 3)]*	GP-implied store double-word immediate	See page 61

Table 4. String Processing Instructions

No.	Arch.	Mnemonic	Instruction	Operation
1	RV32 & RV64	NDS.FFB *Rd, Rs1, Rs2*	Find first Byte	See page 63
2	RV32 & RV64	NDS.FFZMISM *Rd, Rs1, Rs2*	Find first zero or mismatch	See page 68
3	RV32 & RV64	NDS.FFMISM *Rd, Rs1, Rs2*	Find first mismatch	See page 73
4	RV32 & RV64	NDS.FLMISM *Rd, Rs1, Rs2*	Find last mismatch	See page 78

2.2. Andes CoDense Extension (XAndesCoDense)

Table 5. Code Dense Instructions

No.	Arch.	Mnemonic	Instruction	Operation
1	RV32 & RV64	NDS.EXEC.IT *(imm[11:2]<<2)*	Execution on instruction table	`Execute(IT(imm[11:2]);`

2.3. Andes New CoDense Extension (XAndesNewCoDense)

Table 6. New Code Dense Instructions

No.	Arch.	Mnemonic	Instruction	Operation
1	RV32 & RV64	NDS.NEXEC.IT *(imm[11:2]<<2)*	New execution on instruction table. An alias of NDS.EXEC.IT.	`Execute(IT(imm[11:2]);`

2.4. Andes Scalar BFLOAT16 Conversion Extension (XAndesBFHCvt)

Table 7. BFLOAT16 Scalar Conversion Instructions

No.	Arch.	Mnemonic	Instruction	Operation

1	RV32 & RV64	NDS.FCVT.S.BF16 *frd, frs*	Scalar BF16 to 32-bit SP conversion	`frd.H[1] = frs.H[0];`
				`frd.H[0] = 0;`
				`frd = NaN-Boxing(frd.W[0])`

2	RV32 & RV64	NDS.FCVT.BF16.S *frd, frs*	Scalar 32-bit SP to BF16 conversion	`frd.H[0] =`
				`S_SP_TO_BF16(frs.W[0]);`
				`frd = NaN-Boxing(frd.H[0]);`

2.5. Andes Vector BFLOAT16 Conversion Extension (XAndesVBFHCvt)

Table 8. BFLOAT16 Vector Conversion Instructions (for SEW=16 only)

No.	Arch.	Mnemonic	Instruction	Operation
1	RV32 & RV64	`NDS.VFWCVT.S.BF16 vd, vs`	Vector BF16 to 32-bit SP conversion	`vd[i].H[1] = vs[i];` `vd[i].H[0] = 0;`
2	RV32 & RV64	`NDS.VFNCVT.BF16.S vd, vs`	Vector 32-bit SP to BF16 conversion	`vd[i] = V_SP_TO_BF16(vs[i])`

2.6. Andes Vector INT4 Load Extension (XAndesVSIntLoad)

Table 9. INT4 Vector Load Instructions

No.	Arch.	Mnemonic	Instruction	Operation
1	RV32 & RV64	`NDS.VLN8.V vd, (rs1), vm`	Vector signed 4-bit uni-stride load into 8-bit element	See page 99
2	RV32 & RV64	`NDS.VLNU8.V vd, (rs1), vm`	Vector unsigned 4-bit uni-stride load into 8-bit element	See page 101

2.7. Andes Vector Packed FP16 Extension (XAndesVPackFPH)

Table 10. Vector Single-Width Floating-Point Packed Fused Multiply-Add Instructions

No.	Arch.	Mnemonic	Instruction	Operation

1	RV32 & RV64	NDS.VFPMADT.VF *vd, rs1, vs2, vm*	Vector single-width floating-point packed fused multiply-add with top FP16 as multiplicand	See page 103
2	RV32 & RV64	NDS.VFPMADB.VF *vd, rs1, vs2, vm*	Vector single-width floating-point packed fused multiply-add with bottom FP16 as multiplicand	See page 104

2.8. Andes Vector Dot Product Extension (XAndesVDot)

Table 11. Vector Dot Product Instructions

No.	Arch.	Mnemonic	Instruction	Operation
1	RV32 & RV64	NDS.VD4DOTS.VV *vd, vs1, vs2, vm*	Vector signed dot product on 1/4 of SEW	See page 107
2	RV32 & RV64	NDS.VD4DOTU.VV *vd, vs1, vs2, vm*	Vector unsigned dot product on 1/4 of SEW	See page 109
3	RV32 & RV64	NDS.VD4DOTSU.VV *vd, vs1, vs2, vm*	Vector signed and unsigned dot product on 1/4 of SEW	See page 111

2.9. Andes Vector Small INT Handling Extension (XAndesVSIntH)

Table 12. INT4 Vector Load Instructions

No.	Arch.	Mnemonic	Instruction	Operation
1	RV32 & RV64	NDS.VLE4.V *vd, (rs1)*	Vector 4-bit uni-stride load into a vector register	See page 114

Table 13. INT4/INT8 Vector Floating-Point Widening Instructions

No.	Arch.	Mnemonic	Instruction	Operation
1	RV32 & RV64	NDS.VFWCVT.F.N.V *vd, vs, vm*	Vector signed INT4 to SEW FP conversion	See page 116

No.	Arch.	Mnemonic	Instruction	Operation
2	RV32 & RV64	NDS.VFWCVT.F.NU.V *vd, vs, vm*	Vector unsigned INT4 to SEW FP Conversion	See page 119
3	RV32 & RV64	NDS.VFWCVT.F.B.V *vd, vs, vm*	Vector signed INT8 to SEW FP conversion	See page 121
4	RV32 & RV64	NDS.VFWCVT.F.BU.V *vd, vs, vm*	Vector unsigned INT8 to SEW FP Conversion	See page 123

2.10. Andes Vector Quad-Widening Integer Multiply-Add Extension (XAndesVQMac)

Table 14. Quad-Widening Integer Multiply-Add Instructions

No.	Arch.	Mnemonic	Instruction	Operation
1	RV32 & RV64	NDS.VQMACCU.VV *vd, vs1, vs2, vm*	Quad-widening unsigned-integer multiply-add, overwrite addend	`vd[i] = +(vs1[i] * vs2[i]) + vd[i]`
2	RV32 & RV64	NDS.VQMACCU.VX *vd, rs1, vs2, vm*	Quad-widening unsigned-integer multiply-add, overwrite addend	`vd[i] = +(x[rs1] * vs2[i]) + vd[i]`
3	RV32 & RV64	NDS.VQMACC.VV *vd, vs1, vs2, vm*	Quad-widening signed-integer multiply-add, overwrite addend	`vd[i] = +(vs1[i] * vs2[i]) + vd[i]`

No.	Arch.	Mnemonic	Instruction	Operation
4	RV32 & RV64	NDS.VQMACC.VX *vd, rs1, vs2, vm*	Quad-widening signed-integer multiply-add, overwrite addend	`vd[i] = +(x[rs1] * vs2[i]) + vd[i]`
5	RV32 & RV64	NDS.VQMACCSU.VV *vd, vs1, vs2, vm*	Quad-widening signed-unsigned-integer multiply-add, overwrite addend	`vd[i] = +(signed(vs1[i]) * unsigned(vs2[i])) + vd[i]`
6	RV32 & RV64	NDS.VQMACCSU.VX *vd, rs1, vs2, vm*	Quad-widening signed-unsigned-integer multiply-add, overwrite addend	`vd[i] = +(signed(x[rs1]) * unsigned(vs2[i])) + vd[i]`
7	RV32 & RV64	NDS.VQMACCUS.VX *vd, rs1, vs2, vm*	Quad-widening unsigned-signed-integer multiply-add, overwrite addend	`vd[i] = +(unsigned(x[rs1]) * signed(vs2[i])) + vd[i]`

3. Detailed Instruction Description

3.1. Andes Performance Extension (XAndesPerf)

The 32-bit AndeStar V5 extension includes branch instructions, load effective address instructions, GP-relative instructions, and string processing instructions for performance improvement.

3.1.1. NDS.BBC (Branch on Test Bit is Clear/Zero)

Format:

- RV32

31	30	29 25	24 20	19 15	14 12	11 8	7	6 0
imm[10]	BBC 0	imm[9:5]	cimm[4:0]	Rs1	BTBx 111	imm[4:1]	0	Custom-2 1011011

- RV64

31	30	29 25	24 20	19 15	14 12	11 8	7	6 0
imm[10]	BBC 0	imm[9:5]	cimm[4:0]	Rs1	BTBx 111	imm[4:1]	cimm[5]	Custom-2 1011011

Syntax:

- RV32

```
NDS.BBC    Rs1, #cimm[4:0], #imm[10:1]
BBC        Rs1, #cimm[4:0], #imm[10:1]   (deprecated)
```

- RV64

```
NDS.BBC    Rs1, #cimm[5:0], #imm[10:1]
BBC        Rs1, #cimm[5:0], #imm[10:1]   (deprecated)
```

Purpose: Branch on testing a bit when the specified bit is zero.

Description:

- RV32

If the bit position *cimm[4:0]* of the register *Rs1* is zero, this instruction branches to the position of "current PC + sign-extended half-word offset *imm[10:1]*".

- RV64

If the bit position *cimm[5:0]* of the register *Rs1* is zero, this instruction branches to the position of "current PC + sign-extended half-word offset *imm[10:1]*".

Operations:

- RV32

```
if (Rs1[cimm[4:0]]) == 0) {
    tPC = PC + SE(CONCAT(imm[10:1],0[0]));
    PC = tPC
}
```

- RV64

```
if (Rs1[cimm[5:0]]) == 0) {
    tPC = PC + SE(CONCAT(imm[10:1],0[0]));
    PC = tPC
}
```

General exceptions: None

Privilege level: All

3.1.2. NDS.BBS (Branch on Test Bit is Set/Not Zero)

Format:

- RV32

31	30	29 25	24 20	19 15	14 12	11 8	7	6 0
imm[10]	BBS 1	imm[9:5]	cimm[4:0]	Rs1	BTBx 111	imm[4:1]	0	Custom-2 1011011

- RV64

31	30	29 25	24 20	19 15	14 12	11 8	7	6 0
imm[10]	BBS 1	imm[9:5]	cimm[4:0]	Rs1	BTBx 111	imm[4:1]	cimm[5]	Custom-2 1011011

Syntax:

- RV32

```
NDS.BBS  Rs1, #cimm[4:0], #imm[10:1]
BBS      Rs1, #cimm[4:0], #imm[10:1]   (deprecated)
```

- RV64

```
NDS.BBS  Rs1, #cimm[5:0], #imm[10:1]
BBS      Rs1, #cimm[5:0], #imm[10:1]   (deprecated)
```

Purpose: Branch on testing a bit when the specified bit is not zero.

Description:

- RV32

If the bit position *cimm[4:0]* of the register *Rs1* is not zero, this instruction branches to the position of "current PC + sign-extended half-word offset *imm[10:1]*".

- RV64

If the bit position *cimm[5:0]* of the register *Rs1* is not zero, this instruction branches to the position of "current PC + sign-extended half-word offset *imm[10:1]*".

Operations:

- RV32

```
if (Rs1[cimm[4:0]]) != 0) {
    tPC = PC + SE(CONCAT(imm[10:1],0[0]));
    PC = tPC
}
```

- RV64

```
if (Rs1[cimm[5:0]]) != 0) {
    tPC = PC + SE(CONCAT(imm[10:1],0[0]));
    PC = tPC
}
```

General exceptions: None

Privilege level: All

3.1.3. NDS.BEQC (Branch on Equal to Constant)

Format:

31	30	29 25	24 20	19 15	14 12	11 8	7	6 0
imm[10]	cimm[6]	imm[9:5]	cimm[4:0]	Rs1	BEQC 101	imm[4:1]	cimm[5]	Custom-2 1011011

Syntax: NDS.BEQC *Rs1, #cimm[6:0], #imm[10:1]*

BEQC *Rs1, #cimm[6:0], #imm[10:1]* (deprecated)

Purpose: Branch on equal comparison to a constant.

Description: If the content of the register *Rs1* is equal to the zero-extended constant *cimm[6:0]*, this instruction branches to the position of "current PC + sign-extended half-word offset *imm[10:1]*".

Operations:

```
if (Rs1 == ZE(cimm[6:0]) {
    tPC = PC + SE(CONCAT(imm[10:1],0[0]));
    PC = tPC
}
```

General exceptions: None

Privilege level: All

3.1.4. NDS.BNEC (Branch on Not Equal to Constant)

Format:

31	30	29 25	24 20	19 15	14 12	11 8	7	6 0
imm[10]	cimm[6]	imm[9:5]	cimm[4:0]	Rs1	BEQC 110	imm[4:1]	cimm[5]	Custom-2 1011011

Syntax: NDS.BNEC *Rs1, #cimm[6:0], #imm[10:1]*

BNEC *Rs1, #cimm[6:0], #imm[10:1]* (deprecated)

Purpose: Branch on not-equal comparison to a constant.

Description: If the content of the register *Rs1* is not equal to the zero-extended constant *cimm[6:0]*, this instruction branches to the position of "current PC + sign-extended half-word offset *imm[10:1]*".

Operations:

```
if (Rs1 != ZE(cimm[6:0]) {
    tPC = PC + SE(CONCAT(imm[10:1],0[0]));
    PC = tPC
}
```

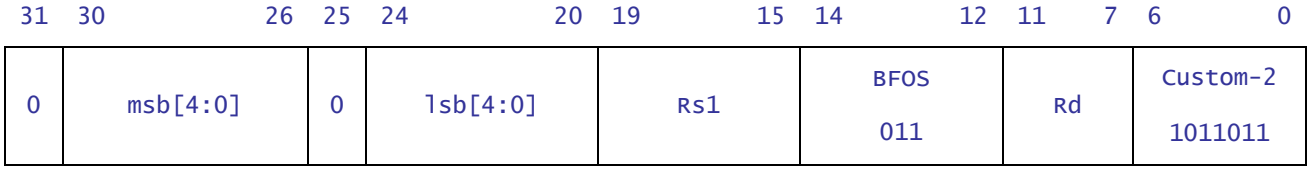
General exceptions: None

Privilege level: All

3.1.5. NDS.BFOS (Sign-Extended Bit Field Operation)

Format:

- RV32



- RV64

Syntax:

- RV32

```
NDS.BFOS   Rd, Rs1, #msb[4:0], #lsb[4:0]
BFOS       Rd, Rs1, #msb[4:0], #lsb[4:0]   (deprecated)
```

- RV64

```
NDS.BFOS   Rd, Rs1, #msb[5:0], #lsb[5:0]
BFOS       Rd, Rs1, #msb[5:0], #lsb[5:0]   (deprecated)
```

Purpose: Perform a sign-extended bit-field extract or insert operation.

Description:

- RV32

This instruction contains three different bit-field operations. If *msb[4:0]* is 0, a sign-extended zero-tailed bit-field insert operation is performed. If *msb[4:0]*< *lsb[4:0]*, another sign-extended bit-field insert operation is performed. If *msb[4:0]*>= *lsb[4:0]*, a sign-extended bit-field extract operation is performed instead.

- RV64

This instruction contains three different bit-field operati*ons.* If *msb[5:0]* is 0, a sign-extended zero-tailed bit-field insert operation is performed. If *msb[5:0]*< *lsb[5:0]*, another sign-extended bit-field insert operation is performed. If *msb[5:0]*>= *lsb[5:0]*, a sign-extended bit-field extract operation is performed instead.

Operations:

- RV32

```
LSB = lsb[4:0]; MSB = msb[4:0];
lsbp1 = lsb[4:0]+1;  msbm1 = msb[4:0]-1;
lsbm1 = lsb[4:0]-1;
if (MSB==0) {
   Rd[LSB]=Rs1[0];
   if (LSB < 31) Rd[31:lsbp1]=REPEAT(Rs1[0]);
   if (LSB > 0) Rd[lsbm1:0]=0;
} else if (MSB<LSB) {
   lenm1 = LSB-MSB;
   Rd[LSB:MSB]=Rs1[lenm1:0];
   if (LSB < 31) Rd[31:lsbp1]=REPEAT(Rs1[lenm1]);
```

```
    Rd[msbm1:0]=0;
} else {   // MSB>=LSB
    lenm1 = MSB-LSB;
    Rd[lenm1:0]=Rs1[MSB:LSB]; Rd[31:(lenm1+1)]=REPEAT(Rs1[MSB]);
}
```

- RV64

```
LSB = lsb[5:0]; MSB = msb[5:0];
lsbp1 = lsb[5:0]+1;  msbm1 = msb[5:0]-1;
lsbm1 = lsb[5:0]-1;
if (MSB==0) {
   Rd[LSB]=Rs1[0];
   if (LSB < 63) Rd[63:lsbp1]=REPEAT(Rs1[0]);
   if (LSB > 0) Rd[lsbm1:0]=0;
}else if (MSB<LSB) {
    lenm1 = LSB-MSB;
    Rd[LSB:MSB]=Rs1[lenm1:0];
    if (LSB < 63) Rd[63:lsbp1]=REPEAT(Rs1[lenm1]);
    Rd[msbm1:0]=0;
} else {   // MSB>=LSB
    lenm1 = MSB-LSB;
    Rd[lenm1:0]=Rs1[MSB:LSB]; Rd[63:(lenm1+1)]=REPEAT(Rs1[MSB]);
```

```
}
```

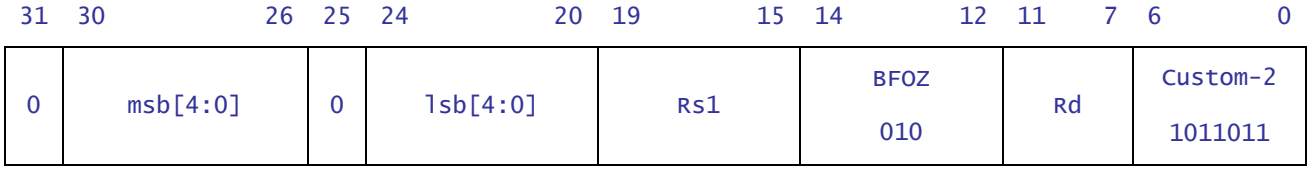
General exceptions: None

Privilege level: All

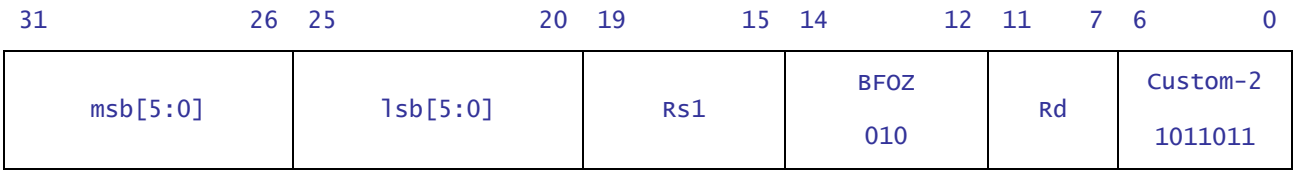
3.1.6. NDS.BFOZ (Zero-Extended Bit Field Operation)

Format:

- RV32



- RV64



Syntax:

- RV32

```
NDS.BFOZ   Rd, Rs1, #msb[4:0], #lsb[4:0]
BFOZ       Rd, Rs1, #msb[4:0], #lsb[4:0]   (deprecated)
```

- RV64

```
NDS.BFOZ   Rd, Rs1, #msb[5:0], #lsb[5:0]
BFOZ       Rd, Rs1, #msb[5:0], #lsb[5:0]   (deprecated)
```

Purpose: Perform a zero-extended bit-field extract or insert operation.

Description:

- RV32

This instruction contains three different bit-field operations. If *msb[4:0]* is 0, a zero-extended zero-tailed bit-field insert operation is performed. If *msb[4:0]*<*lsb[4:0]*, another zero-extended bit-field insert operation is performed. If *msb[4:0]*>=*lsb[4:0]*, a zero-extended bit-field extract operation is performed instead.

- RV64

This instruction contains three different bit-field operations. If *msb[5:0]* is 0, a zero-extended zero-tailed bit-field insert operation is performed. If *msb[5:0]*<*lsb[5:0]*, another zero-extended bit-field insert operation is performed. If *msb[5:0]*>=*lsb[5:0]*, a zero-extended bit-field extract operation is performed instead.

Operations:

- RV32

```
LSB = lsb[4:0]; MSB = msb[4:0];
lsbp1 = lsb[4:0]+1;  msbm1 = msb[4:0]-1;
lsbm1 = lsb[4:0]-1;
if (MSB==0) {
   Rd[LSB]=Rs1[0];
   if (LSB < 31) Rd[31:lsbp1]=0;
   if (LSB > 0) Rd[lsbm1:0]=0;
 }else if (MSB<LSB) {
   lenm1 = LSB-MSB;
   Rd[LSB:MSB]=Rs1[lenm1:0];
```

```
    if (LSB < 31) Rd[31:lsbp1]=0;
    Rd[msbm1:0]=0;
  } else {   // MSB>=LSB
    lenm1 = MSB-LSB;
    Rd[lenm1:0]=Rs1[MSB:LSB]; Rd[31:(lenm1+1)]=0;
  }
```

- RV64

```
  LSB = lsb[5:0]; MSB = msb[5:0];
  lsbp1 = lsb[5:0]+1;  msbm1 = msb[5:0]-1;
  lsbm1 = lsb[5:0]-1;
  if (MSB==0) {
    Rd[LSB]=Rs1[0];
    if (LSB < 63) Rd[63:lsbp1]=0;
    if (LSB > 0) Rd[lsbm1:0]=0;
  }else if (MSB<LSB) {
    lenm1 = LSB-MSB;
    Rd[LSB:MSB]=Rs1[lenm1:0];
    if (LSB < 63) Rd[63:lsbp1]=0;
    Rd[msbm1:0]=0;
  } else {   // MSB>=LSB
    lenm1 = MSB-LSB;
```

```
        Rd[lenm1:0]=Rs1[MSB:LSB]; Rd[63:(lenm1+1)]=0;
    }
```

General exceptions: None

Privilege level: All

3.1.7. NDS.LEA.H (Load Effective Half-Word Address)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
LEA.H 0000101	Rs2	Rs1	000	Rd	Custom-2 1011011

Syntax: NDS.LEA.H *Rd, Rs1, Rs2*

LEA.H *Rd, Rs1, Rs2* (deprecated)

Purpose: Add a base register with a half-word-aligned offset from an offset register.

Description: This instruction adds the content of the register *Rs1* with the 1-bit left-shifted content of the register *Rs2* and writes the result to the register *Rd*.

Operations:

```
Rd = Rs1 + Rs2*2;
```

General exceptions: None

Privilege level: All

3.1.8. NDS.LEA.W (Load Effective Word Address)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
LEA.W 0000110	Rs2	Rs1	000	Rd	Custom-2 1011011

Syntax: NDS.LEA.W *Rd, Rs1, Rs2*

LEA.W *Rd, Rs1, Rs2* (deprecated)

Purpose: Add a base register with a word-aligned offset from an offset register.

Description: This instruction adds the content of the register *Rs1* with the 2-bit left-shifted content of the register *Rs2* and writes the result to the register *Rd*.

Operations:

```
Rd = Rs1 + Rs2*4;
```

General exceptions: None

Privilege level: All

3.1.9. NDS.LEA.D (Load Effective Double-Word Address)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
LEA.D 0000111	Rs2	Rs1	000	Rd	Custom-2 1011011

Syntax: NDS.LEA.D *Rd, Rs1, Rs2*

LEA.D *Rd, Rs1, Rs2* (deprecated)

Purpose: Add a base register with a double-word-aligned offset from an offset register.

Description: This instruction adds the content of the register *Rs1* with the 3-bit left-shifted content of the register *Rs2* and writes the result to the register *Rd*.

Operations:

```
Rd = Rs1 + Rs2*8;
```

General exceptions: None

Privilege level: All

3.1.10. NDS.LEA.B.ZE (Load Effective Byte Address from Unsigned 32-Bit Offset)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
LEA.B.ZE 0001000	Rs2	Rs1	000	Rd	Custom-2 1011011

Syntax: NDS.LEA.B.ZE *Rd, Rs1, Rs2*

LEA.B.ZE *Rd, Rs1, Rs2* (deprecated)

Purpose: Add a base register with an unsigned 32-bit byte offset from an offset register.

Description: For RV64 only, this instruction adds the content of the register *Rs1* with the zero-extended [31:0] content of the register *Rs2* and writes the result to the register *Rd*.

Operations:

```
Rd = Rs1 + ZE32(Rs2[31:0]);
```

General exceptions: None

Privilege level: All

3.1.11. NDS.LEA.H.ZE (Load Effective Half-word Address from Unsigned 32-Bit Offset)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
LEA.H.ZE 0001001	Rs2	Rs1	000	Rd	Custom-2 1011011

Syntax: `NDS.LEA.H.ZE` *Rd, Rs1, Rs2*

`LEA.H.ZE` *Rd, Rs1, Rs2* (deprecated)

Purpose: Add a base register with an unsigned 32-bit half-word offset from an offset register.

Description: For RV64 only, this instruction adds the content of the register *Rs1* with the 1-bit left-shifted zero-extended `[31:0]` content of the register *Rs2* and writes the result to the register *Rd*.

Operations:

```
Rd = Rs1 + ZE32(Rs2[31:0])*2;
```

General exceptions: None

Privilege level: All

3.1.12. NDS.LEA.W.ZE (Load Effective Word Address from Unsigned 32-Bit Offset)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
LEA.W.ZE 0001010	Rs2	Rs1	000	Rd	Custom-2 1011011

Syntax: NDS.LEA.W.ZE *Rd, Rs1, Rs2*

LEA.W.ZE *Rd, Rs1, Rs2* (deprecated)

Purpose: Add a base register with an unsigned 32-bit word offset from an offset register.

Description: For RV64 only, this instruction adds the content of the register *Rs1* with the 2-bit left-shifted zero-extended [31:0] content of the register *Rs2* and writes the result to the register *Rd*.

Operations:

```
Rd = Rs1 + ZE32(Rs2[31:0])*4;
```

General exceptions: None

Privilege level: All

3.1.13. NDS.LEA.D.ZE (Load Effective Double-Word Address from Unsigned 32-Bit Offset)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
LEA.D.ZE 0001011	Rs2	Rs1	000	Rd	Custom-2 1011011

Syntax: NDS.LEA.D.ZE *Rd, Rs1, Rs2*

LEA.D.ZE *Rd, Rs1, Rs2* (deprecated)

Purpose: Add a base register with an unsigned 32-bit double-word offset from an offset register.

Description: For RV64 only, this instruction adds the content of the register *Rs1* with the 3-bit left-shifted zero-extended [31:0] content of the register *Rs2* and writes the result to the register *Rd*.

Operations:

```
Rd = Rs1 + ZE32(Rs2[31:0])*8;
```

General exceptions: None

Privilege level: All

3.1.14. NDS.ADDIGP (GP-Implied Add Immediate)

Format:

31 30 21	20	19 17	16 15	14	13 12	11 7	6 0
imm17	imm[10:1]	imm11	imm[14:12]	imm[16:15]	imm0	ADDIGP 01	Rd	Custom-0 0001011

Syntax: NDS.ADDIGP *Rd, imm[17:0]*

ADDIGP *Rd, imm[17:0]* (deprecated)

Purpose: Add the content of the implied global pointer (GP) register x3 with a signed constant.

Description: This instruction adds the content of the GP register x3 with the sign-extended *imm[17:0]* offset value that covers a 256KiB range relative to the location pointed to by the GP register and writes the result to the register *Rd*.

Operations:

```
Rd = x3 + SE(imm[17:0]);
```

General exceptions: None

Privilege level: All

3.1.15. NDS.LBGP (GP-Implied Load Byte Signed Immediate)

Format:

31	30 21	20	19 17	16 15	14	13 12	11 7	6 0
imm17	imm[10:1]	imm11	imm[14:12]	imm[16:15]	imm0	LBGP 00	Rd	Custom-0 0001011

Syntax: NDS.LBGP *Rd, [+ imm[17:0]]*

LBGP *Rd, [+ imm[17:0]]* (deprecated)

Purpose: Load a sign-extended 8-bit byte from the memory into a general register.

Description: This instruction loads a sign-extended byte from a memory location into the general register *Rd*. The address of the memory location is specified by the implied GP register x3 plus a sign-extended *imm[17:0]* offset that covers a 256KiB range relative to the location pointed to by the GP register.

Operations:

```
Vaddr = x3 + Sign_Extend(imm[17:0]);
(PAddr, Attributes) = Address_Translation(Vaddr);
Excep_status = Page_Exception(Attributes, POM, LOAD);
If (Excep_status == NO_EXCEPTION) {
   Bdata(7,0) = Load_Memory(PAddr, BYTE, Attributes);
   Rd = Sign_Extend(Bdata(7,0));
```

```
} else {
   Generate_Exception(Excep_status);
}
```

General exceptions: TLB fill, non-leaf PTE not present, leaf PTE not present, read protection, page modified, access bit, load access fault, bus error, ECC/parity error

Privilege level: All

3.1.16. NDS.LBUGP (GP-Implied Load Byte Unsigned Immediate)

Format:

31	30 21	20	19 17	16 15	14	13 12	11 7	6 0
imm17	imm[10:1]	imm11	imm[14:12]	imm[16:15]	imm0	LBUGP 10	Rd	Custom-0 0001011

Syntax: NDS.LBUGP *Rd, [+ imm[17:0]]*

LBUGP *Rd, [+ imm[17:0]]* (deprecated)

Purpose: Load a zero-extended 8-bit byte from the memory into a general register.

Description: This instruction loads a zero-extended byte from a memory location into the general register *Rd*. The address of the memory location is specified by the implied GP register x3 plus a sign-extended *imm[17:0]* offset that covers a 256KiB range relative to the location pointed to by the GP register.

Operations:

```
Vaddr = x3 + Sign_Extend(imm[17:0]);
(PAddr, Attributes) = Address_Translation(Vaddr);
Excep_status = Page_Exception(Attributes, POM, LOAD);
If (Excep_status == NO_EXCEPTION) {
   Bdata(7,0) = Load_Memory(PAddr, BYTE, Attributes);
   Rd = Zero_Extend(Bdata(7,0));
```

```
} else {
   Generate_Exception(Excep_status);
}
```

General exceptions: TLB fill, non-leaf PTE not present, leaf PTE not present, read protection, page modified, access bit, load access fault, bus error, ECC/parity error

Privilege level: All

3.1.17. NDS.LHGP (GP-Implied Load Half-Word Signed Immediate)

Format:

31 30 21	20 19 17	16 15	14 12	11 7	6 0
imm17	imm[10:1]	imm11	imm[14:12]	imm[16:15]	LHGP 001	Rd	Custom-1 0101011

Syntax: NDS.LHGP *Rd, [+ (imm[17:1] << 1)]*

LHGP *Rd, [+ (imm[17:1] << 1)]* (deprecated)

Purpose: Load a sign-extended 16-bit half-word from the memory into a general register.

Description: This instruction loads a sign-extended half-word from the memory into the general register *Rd*. The address of the memory location is specified by the implied GP register x3 plus a sign-extended half-word *imm[17:1]* offset that covers a 256KiB range relative to the location pointed to by the GP register.

Operations:

```
Vaddr = x3 + Sign_Extend((imm[17:1]<<1));
(PAddr, Attributes) = Address_Translation(Vaddr);
Excep_status = Page_Exception(Attributes, POM, LOAD);
If (Excep_status == NO_EXCEPTION) {
   Hdata(15,0) = Load_Memory(PAddr, HWORD, Attributes);
```

```
        Rd = Sign_Extend(Hdata(15,0));
    } else {
        Generate_Exception(Excep_status);
    }
```

General exceptions: TLB fill, non-leaf PTE not present, leaf PTE not present, read protection, page modified, access bit, load access fault, bus error, ECC/parity error

Privilege level: All

3.1.18. NDS.LHUGP (GP-Implied Load Half-Word Unsigned Immediate)

Format:

31 30 21	20 19 17	16 15	14 12	11 7	6 0
imm17	imm[10:1]	imm11	imm[14:12]	imm[16:15]	LHUGP 101	Rd	Custom-1 0101011

Syntax: NDS.LHUGP *Rd, [+ (imm[17:1] << 1)]*

LHUGP *Rd, [+ (imm[17:1] << 1)]* (deprecated)

Purpose: Load a zero-extended 16-bit half-word from the memory into a general register.

Description: This instruction loads a zero-extended half-word from a memory location into the general register *Rd*. The address of the memory location is specified by the implied GP register x3 plus a sign-extended half-word *imm[17:1]* offset that covers a 256KiB range relative to the location pointed to by the GP register.

Operations:

```
Vaddr = x3 + Sign_Extend((imm[17:1]<<1));
(PAddr, Attributes) = Address_Translation(Vaddr);
Excep_status = Page_Exception(Attributes, POM, LOAD);
If (Excep_status == NO_EXCEPTION) {
   Hdata(15,0) = Load_Memory(PAddr, HWORD, Attributes);
```

```
        Rd = Zero_Extend(Hdata(15,0));
    } else {
        Generate_Exception(Excep_status);
    }
```

General exceptions: TLB fill, non-leaf PTE not present, leaf PTE not present, read protection, page modified, access bit, load access fault, bus error, ECC/parity error

Privilege level: All

3.1.19. NDS.LWGP (GP-Implied Load Word Signed Immediate)

Format:

31 30 22	21	20	19 17	16 15	14 12	11 7	6 0
imm18	imm[10:2]	imm17	imm11	imm[14:12]	imm[16:15]	LWGP 010	Rd	Custom-1 0101011

Syntax: NDS.LWGP *Rd, [+ (imm[18:2] <<2)]*

LWGP *Rd, [+ (imm[18:2] <<2)]* (deprecated)

Purpose: Load a sign-extended 32-bit word from the memory into a general register.

Description: This instruction loads a sign-extended word from a memory location into the general register *Rd*. The address of the memory location is specified by the implied GP register x3 plus a sign-extended word *imm[18:2]* offset that covers a 512KiB range relative to the location pointed to by the GP register.

Operations:

```
Vaddr = x3 + Sign_Extend((imm[18:2]<<2));
(PAddr, Attributes) = Address_Translation(Vaddr);
Excep_status = Page_Exception(Attributes, POM, LOAD);
If (Excep_status == NO_EXCEPTION) {
   Wdata(31,0) = Load_Memory(PAddr, WORD, Attributes);
```

```
        Rd = Sign_Extend(Wdata(31,0));
    } else {
        Generate_Exception(Excep_status);
    }
```

General exceptions: TLB fill, non-leaf PTE not present, leaf PTE not present, read protection, page modified, access bit, load access fault, bus error, ECC/parity error

Privilege level: All

3.1.20. NDS.LWUGP (GP-Implied Load Word Unsigned Immediate)

Format:

31 30 22	21	20	19 17	16 15	14 12	11 7	6 0
imm18	imm[10:2]	imm17	imm11	imm[14:12]	imm[16:15]	LWUGP 110	Rd	Custom-1 0101011

Syntax: NDS.LWUGP *Rd, [+ (imm[18:2] <<2)]*

LWUGP *Rd, [+ (imm[18:2] <<2)]* (deprecated)

Purpose: Load a zero-extended 32-bit word from the memory into a general register.

Description: For RV64 only, this instruction loads a zero-extended word from a memory location into the general register *Rd*. The address of the memory location is specified by the implied GP register x3 plus a sign-extended word *imm[18:2]* offset that covers a 512KiB range relative to the location pointed to by the GP register.

Operations:

```
Vaddr = x3 + Sign_Extend((imm[18:2]<<2));
(PAddr, Attributes) = Address_Translation(Vaddr);
Excep_status = Page_Exception(Attributes, POM, LOAD);
If (Excep_status == NO_EXCEPTION) {
   Wdata(31,0) = Load_Memory(PAddr, WORD, Attributes);
```

```
        Rd = Zero_Extend(Wdata(31,0));
    } else {
        Generate_Exception(Excep_status);
    }
```

General exceptions: TLB fill, non-leaf PTE not present, leaf PTE not present, read protection, page modified, access bit, load access fault, bus error, ECC/parity error

Privilege level: All

3.1.21. NDS.LDGP (GP-Implied Load Double-Word Immediate)

Format:

31 30 23	22 21	20	19 17	16 15	14 12	11 7	6 0
imm19	imm[10:3]	imm[18:17]	imm11	imm[14:12]	imm[16:15]	LDGP 011	Rd	Custom-1 0101011

Syntax: NDS.LDGP *Rd, [+ (imm[19:3] <<3)]*

LDGP *Rd, [+ (imm[19:3] <<3)]* (deprecated)

Purpose: Load a 64-bit double-word from the memory into a general register.

Description: For RV64 only, this instruction loads a double-word from a memory location into the general register *Rd*. The address of the memory location is specified by the implied GP register x3 plus a sign-extended double-word *imm[19:3]* offset that covers a 1024KiB range relative to the location pointed to by the GP register.

Operations:

```
Vaddr = x3 + Sign_Extend((imm[19:3]<<3));
(PAddr, Attributes) = Address_Translation(Vaddr);
Excep_status = Page_Exception(Attributes, POM, LOAD);
If (Excep_status == NO_EXCEPTION) {
   Dwdata(63,0) = Load_Memory(PAddr, DWORD, Attributes);
```

```
        Rd = Sign_Extend(Dwdata(63,0));
    } else {
        Generate_Exception(Excep_status);
    }
```

General exceptions: TLB fill, non-leaf PTE not present, leaf PTE not present, read protection, page modified, access bit, load access fault, bus error, ECC/parity error

Privilege level: All

3.1.22. NDS.SBGP (GP-Implied Store Byte Immediate)

Format:

31 30 25	24 20	19 17	16 15	14	13 12	11 8	7	6 0
imm17	imm[10:5]	Rs2	imm[14:12]	imm[16:15]	imm0	SBGP 11	imm[4:1]	imm11	Custom-0 0001011

Syntax: NDS.SBGP *Rs2, [+ imm[17:0]]*

SBGP *Rs2, [+ imm[17:0]]* (deprecated)

Purpose: Store an 8-bit byte from a general register into a memory location.

Description: This instruction stores the least-significant 8-bit byte in the general register *Rs2* to a memory address specified by the implied GP register x3 plus a sign-extended *imm[17:0]* offset. The offset value covers a 256KiB range relative to the location pointed to by the GP register.

Operations:

```
Vaddr = x3 + Sign_Extend(imm[17:0]);
(PAddr, Attributes) = Address_Translation(Vaddr);
Excep_status = Page_Exception(Attributes, POM, STORE);
If (Excep_status == NO_EXCEPTION) {
   Store_Memory(PAddr, BYTE, Attributes, Rs2(7,0));
} else {
```

```
        Generate_Exception(Excep_status);
    }
```

General exceptions: TLB fill, non-leaf PTE not present, leaf PTE not present, read protection, page modified, access bit, store access fault, bus error, ECC/parity error

Privilege level: All

3.1.23. NDS.SHGP (GP-Implied Store Half-Word Immediate)

Format:

31 30 25	24 20	19 17	16 15	14 12	11 8	7	6 0
imm17	imm[10:5]	Rs2	imm[14:12]	imm[16:15]	SHGP 000	imm[4:1]	imm11	Custom-1 0101011

Syntax: NDS.SHGP *Rs2, [+ (imm[17:1] << 1)]*

SHGP *Rs2, [+ (imm[17:1] << 1)]* (deprecated)

Purpose: Store a 16-bit half-word from a general register into a memory location.

Description: This instruction stores the least-significant 16-bit half-word in the general register *Rs2* to a memory address specified by the implied GP register x3 plus a sign-extended half-word *imm[17:1]* offset. The offset value covers a 256KiB range relative to the location pointed to by the GP register.

Operations:

```
Vaddr = x3 + Sign_Extend((imm[17:1]<<1));
(PAddr, Attributes) = Address_Translation(Vaddr);
Excep_status = Page_Exception(Attributes, POM, STORE);
If (Excep_status == NO_EXCEPTION) {
   Store_Memory(PAddr, HWORD, Attributes, Rs2(15,0));
} else {
```

```
        Generate_Exception(Excep_status);
    }
```

General exceptions: TLB fill, non-leaf PTE not present, leaf PTE not present, read protection, page modified, access bit, store access fault, bus error, ECC/parity error

Privilege level: All

3.1.24. NDS.SWGP (GP-Implied Store Word Immediate)

Format:

31 30 25	24 20	19 17	16 15	14 12	11 9	8 7	6 0
imm18	imm[10:5]	Rs2	imm[14:12]	imm[16:15]	SWGP 100	imm[4:2]	imm17	imm11	Custom-1 0101011

Syntax: NDS.SWGP *Rs2, [+ (imm[18:2] << 2)]*

SWGP *Rs2, [+ (imm[18:2] << 2)]* (deprecated)

Purpose: Store a 32-bit word from a general register into a memory location.

Description: This instruction store the least-significant 32-bit word in the general register *Rs2* to a memory address specified by the implied GP register x3 plus a sign-extended word *imm[18:2]* offset. The offset value covers a 512KiB range relative to the location pointed to by the GP register.

Operations:

```
Vaddr = x3 + Sign_Extend((imm[18:2]<<2));
(PAddr, Attributes) = Address_Translation(Vaddr);
Excep_status = Page_Exception(Attributes, POM, STORE);
If (Excep_status == NO_EXCEPTION) {
   Store_Memory(PAddr, WORD, Attributes, Rs2(31,0));
} else {
```

```
        Generate_Exception(Excep_status);
    }
```

General exceptions: TLB fill, non-leaf PTE not present, leaf PTE not present, read protection, page modified, access bit, store access fault, bus error, ECC/parity error

Privilege level: All

3.1.25. NDS.SDGP (GP-Implied Store Double-Word Immediate)

Format:

31 30 25	24 20	19 17	16 15	14 12	11 10	9 8	7	6 0
imm19	imm[10:5]	Rs2	imm[14:12]	imm[16:15]	SWGP 111	imm[4:3]	imm[18:17]	imm11	Custom-1 0101011

Syntax: NDS.SDGP *Rs2, [+ (imm[19:3] << 3)]*

SDGP *Rs2, [+ (imm[19:3] << 3)]* (deprecated)

Purpose: Store a 64-bit double-word from a general register into a memory location.

Description: For RV64 only, the instruction stores the 64-bit double-word in the general register *Rs2* to a memory address specified by the implied GP register x3 plus a sign-extended double-word *imm[19:3]* offset. The offset value covers a 1024KiB range relative to the location pointed to by the GP register.

Operations:

```
Vaddr = x3 + Sign_Extend((imm[19:3]<<3));
(PAddr, Attributes) = Address_Translation(Vaddr);
Excep_status = Page_Exception(Attributes, POM, STORE);
If (Excep_status == NO_EXCEPTION) {
   Store_Memory(PAddr, DWORD, Attributes, Rs2(63,0));
} else {
```

```
        Generate_Exception(Excep_status);
    }
```

General exceptions: TLB fill, non-leaf PTE not present, leaf PTE not present, read protection, page modified, access bit, store access fault, bus error, ECC/parity error

Privilege level: All

3.1.26. NDS.FFB (Find First Byte)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
FFB 0010000	Rs2	Rs1	000	Rd	Custom-2 1011011

Syntax: NDS.FFB *Rd, Rs1, Rs2*

FFB *Rd, Rs1, Rs2* (deprecated)

Purpose: Find the first byte in a register that matches a value in another register.

Description: Each byte in the register *Rs1* is compared with the values in *Rs2[7:0]*. If any matching byte is found, a non-zero position indication of the first matching byte is written to the register *Rd* and the result is data-endian dependent. If no matching byte is found, a zero is written to the register *Rd*.

Operations:

- RV32

```
Match1 = (Rs1[7:0] == Rs2[7:0]);   Match2 = (Rs1[15:8] == Rs2[7:0]);
Match3 = (Rs1[23:16] == Rs2[7:0]); Match4 = (Rs1[31:24] == Rs2[7:0]);
found =  Match1 || Match2 || Match3 || Match4;
If (!found) {
   Rd = 0;
```

```
} else if (“Little Endian”) {
   If (Match1) {
      Rd = -4;
   } else if (Match2) {
      Rd = -3;
   } else if (Match3) {
      Rd = -2;
   } else {
      Rd = -1;
   }
} else { // “Big Endian”
   If (Match4) {
      Rd = -4;
   } else if (Match3) {
      Rd = -3;
   } else if (Match2) {
      Rd = -2;
   } else {
      Rd = -1;
   }
}
```

- RV64

```
Match1 = (Rs1[7:0] == Rs2[7:0]);   Match2 = (Rs1[15:8] == Rs2[7:0]);

Match3 = (Rs1[23:16] == Rs2[7:0]); Match4 = (Rs1[31:24] == Rs2[7:0]);

Match5 = (Rs1[39:32] == Rs2[7:0]); Match6 = (Rs1[47:40] == Rs2[7:0]);

Match7 = (Rs1[55:48] == Rs2[7:0]); Match8 = (Rs1[63:56] == Rs2[7:0]);

found =  Match1 || Match2 || Match3 || Match4 ||
          Match5 || Match6 || Match7 || Match8;

If (!found) {

   Rd = 0;

} else if ("Little Endian") {

   If (Match1) {

      Rd = -8;

   } else if (Match2) {

      Rd = -7;

   } else if (Match3) {

      Rd = -6;

   } else if (Match4) {

      Rd = -5;

   } else if (Match5) {

      Rd = -4;
```

```
      } else if (Match6) {
         Rd = -3;
      } else if (Match7) {
         Rd = -2;
      } else {
         Rd = -1;
      }
   } else { // “Big Endian”
      If (Match8) {
         Rd = -8;
      } else if (Match7) {
         Rd = -7;
      } else if (Match6) {
         Rd = -6;
      } else if (Match5) {
         Rd = -5;
      } else if (Match4) {
         Rd = -4;
      } else if (Match3) {
         Rd = -3;
      } else if (Match2) {
```

```
            Rd = -2;
        } else {
            Rd = -1;
        }
    }
```

General exceptions: None

Privilege level: All

3.1.27. NDS.FFZMISM (Find First Zero or Mis-Match)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
FFZMISM 0010001	Rs2	Rs1	000	Rd	Custom-2 1011011

Syntax: NDS.FFZMISM *Rd, Rs1, Rs2*

FFZMISM *Rd, Rs1, Rs2* (deprecated)

Purpose: Find the first byte in a register that is zero or fails a corresponding byte comparison.

Description: Each byte in the register *Rs1* is compared with each corresponding byte in the register *Rs2*. If any zero byte or mis-matching byte is found, a non-zero position indication of the first such byte is written to the register *Rd* and the result is data-endian dependent. If no such byte is found, a zero is written to the register *Rd*.

Operations:

- RV32

```
F1 = (Rs1[7:0] == 0)   || (Rs1[7:0] != Rs2[7:0]);
F2 = (Rs1[15:8] == 0)  || (Rs1[15:8] != Rs2[15:8]);
F3 = (Rs1[23:16] == 0) || (Rs1[23:16] != Rs2[23:16]);
F4 = (Rs1[31:24] == 0) || (Rs1[31:24] != Rs2[31:24]);
found =  F1 || F2 || F3 || F4;
```

```
If (!found) {
   Rd = 0;
} else if (“Little Endian”) {
   If (F1) {
      Rd = -4;
   } else if (F2) {
      Rd = -3;
   } else if (F3) {
      Rd = -2;
   } else {
      Rd = -1;
   }
} else { // “Big Endian”
   If (F4) {
      Rd = -4;
   } else if (F3) {
      Rd = -3;
   } else if (F2) {
      Rd = -2;
   } else {
      Rd = -1;
```

```
    }
}
```

- RV64

```
F1 = (Rs1[7:0] == 0)   || (Rs1[7:0] != Rs2[7:0]);
F2 = (Rs1[15:8] == 0)  || (Rs1[15:8] != Rs2[15:8]);
F3 = (Rs1[23:16] == 0) || (Rs1[23:16] != Rs2[23:16]);
F4 = (Rs1[31:24] == 0) || (Rs1[31:24] != Rs2[31:24]);
F5 = (Rs1[39:32] == 0) || (Rs1[39:32] != Rs2[39:32]);
F6 = (Rs1[47:40] == 0) || (Rs1[47:40] != Rs2[47:40]);
F7 = (Rs1[55:48] == 0) || (Rs1[55:48] != Rs2[55:48]);
F8 = (Rs1[63:56] == 0) || (Rs1[63:56] != Rs2[63:56]);
found =  F1 || F2 || F3 || F4 || F5 || F6 || F7 || F8;
If (!found) {
   Rd = 0;
} else if (“Little Endian”) {
   If (F1) {
      Rd = -8;
   } else if (F2) {
      Rd = -7;
   } else if (F3) {
```

```
        Rd = -6;
    } else if (F4) {
        Rd = -5;
    } else if (F5) {
        Rd = -4;
    } else if (F6) {
        Rd = -3;
    } else if (F7) {
        Rd = -2;
    } else {
        Rd = -1;
    }
} else { // “Big Endian”
    If (F8) {
        Rd = -8;
    } else if (F7) {
        Rd = -7;
    } else if (F6) {
        Rd = -6;
    } else if (F5) {
        Rd = -5;
```

```
        } else if (F4) {
           Rd = -4;
        } else if (F3) {
           Rd = -3;
        } else if (F2) {
           Rd = -2;
        } else {
           Rd = -1;
        }
    }
```

General exceptions: None

Privilege level: All

3.1.28. NDS.FFMISM (Find First Mis-Match)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
FFMISM 0010010	Rs2	Rs1	000	Rd	Custom-2 1011011

Syntax: NDS.FFMISM *Rd, Rs1, Rs2*

FFMISM *Rd, Rs1, Rs2* (deprecated)

Purpose: Find the first byte in a register that fails a corresponding byte comparison.

Description: Each byte in the register *Rs1* is compared with each corresponding byte in the register *Rs2*. If any mismatching byte is found, a non-zero position indication of the first mismatching byte is written to the register *Rd* and the result is data-endian dependent. If no mismatching byte is found, a zero is written to the register *Rd*.

Operations:

- RV32

```
MisM1 = (Rs1[7:0] != Rs2[7:0]);
MisM2 = (Rs1[15:8] != Rs2[15:8]);
MisM3 = (Rs1[23:16] != Rs2[23:16]);
MisM4 = (Rs1[31:24] != Rs2[31:24]);
found =  MisM1 || MisM2 || MisM3 || MisM4;
```

```
If (!found) {
   Rd = 0;
} else if (“Little Endian”) {
   If (MisM1) {
      Rd = -4;
   } else if (MisM2) {
      Rd = -3;
   } else if (MisM3) {
      Rd = -2;
   } else {
      Rd = -1;
   }
} else { // “Big Endian”
   If (MisM4) {
      Rd = -4;
   } else if (MisM3) {
      Rd = -3;
   } else if (MisM2) {
      Rd = -2;
   } else {
      Rd = -1;
```

```
    }
}
```

- RV64

```
MisM1 = (Rs1[7:0] != Rs2[7:0]);
MisM2 = (Rs1[15:8] != Rs2[15:8]);
MisM3 = (Rs1[23:16] != Rs2[23:16]);
MisM4 = (Rs1[31:24] != Rs2[31:24]);
MisM5 = (Rs1[39:32] != Rs2[39:32]);
MisM6 = (Rs1[47:40] != Rs2[47:40]);
MisM7 = (Rs1[55:48] != Rs2[55:48]);
MisM8 = (Rs1[63:56] != Rs2[63:56]);
found =  MisM1 || MisM2 || MisM3 || MisM4 || MisM5 || MisM6 || MisM7 ||
MisM8;
If (!found) {
   Rd = 0;
} else if (“Little Endian”) {
   If (MisM1) {
      Rd = -8;
   } else if (MisM2) {
      Rd = -7;
```

```
      } else if (MisM3) {
         Rd = -6;
      } else if (MisM4) {
         Rd = -5;
      } else if (MisM5) {
         Rd = -4;
      } else if (MisM6) {
         Rd = -3;
      } else if (MisM7) {
         Rd = -2;
      } else {
         Rd = -1;
      }
   } else { // “Big Endian”
      If (MisM8) {
         Rd = -8;
      } else if (MisM7) {
         Rd = -7;
      } else if (MisM6) {
         Rd = -6;
      } else if (MisM5) {
```

```
            Rd = -5;
        } else if (MisM4) {
            Rd = -4;
        } else if (MisM3) {
            Rd = -3;
        } else if (MisM2) {
            Rd = -2;
        } else {
            Rd = -1;
        }
    }
```

General exceptions: None

Privilege level: All

3.1.29. NDS.FLMISM (Find Last Mis-Match)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
FLMISM 0010011	Rs2	Rs1	000	Rd	Custom-2 1011011

Syntax: NDS.FLMISM *Rd, Rs1, Rs2*

FLMISM *Rd, Rs1, Rs2* (deprecated)

Purpose: Find the last byte in a register that fails a corresponding byte comparison.

Description: Each byte in the register *Rs1* is compared with each corresponding byte in the register *Rs2*. If any mismatching byte is found, a non-zero position indication of the last mismatching byte is written to the register *Rd* and the result is data-endian dependent. If no mismatching byte is found, a zero is written to the register *Rd*.

Operations:

- RV32

```
MisM1 = (Rs1[7:0] != Rs2[7:0]);
MisM2 = (Rs1[15:8] != Rs2[15:8]);
MisM3 = (Rs1[23:16] != Rs2[23:16]);
MisM4 = (Rs1[31:24] != Rs2[31:24]);
found =  MisM1 || MisM2 || MisM3 || MisM4;
```

```
If (!found) {
   Rd = 0;
} else if (“Little Endian”) {
   If (MisM4) {
      Rd = -1;
   } else if (MisM3) {
      Rd = -2;
   } else if (MisM2) {
      Rd = -3;
   } else {
      Rd = -4;
   }
} else { // Big Endian
   If (MisM1) {
      Rd = -1;
   } else if (MisM2) {
      Rd = -2;
   } else if (MisM3) {
      Rd = -3;
   } else {
      Rd = -4;
```

```
    }
}
```

- RV64

```
MisM1 = (Rs1[7:0] != Rs2[7:0]);
MisM2 = (Rs1[15:8] != Rs2[15:8]);
MisM3 = (Rs1[23:16] != Rs2[23:16]);
MisM4 = (Rs1[31:24] != Rs2[31:24]);
MisM5 = (Rs1[39:32] != Rs2[39:32]);
MisM6 = (Rs1[47:40] != Rs2[47:40]);
MisM7 = (Rs1[55:48] != Rs2[55:48]);
MisM8 = (Rs1[63:56] != Rs2[63:56]);
found =  MisM1 || MisM2 || MisM3 || MisM4 || MisM5 || MisM6 || MisM7 ||
MisM8;
If (!found) {
   Rd = 0;
} else if (“Little Endian”) {
   If (MisM8) {
      Rd = -1;
   } else if (MisM7) {
      Rd = -2;
```

```
      } else if (MisM6) {
         Rd = -3;
      } else if (MisM5) {
         Rd = -4;
      } else if (MisM4) {
         Rd = -5;
      } else if (MisM3) {
         Rd = -6;
      } else if (MisM2) {
         Rd = -7;
      } else {
         Rd = -8;
      }
   } else { // Big Endian
      If (MisM1) {
         Rd = -1;
      } else if (MisM2) {
         Rd = -2;
      } else if (MisM3) {
         Rd = -3;
      } else if (MisM4) {
```

```
            Rd = -4;
        } else if (MisM5) {
            Rd = -5;
        } else if (MisM6) {
            Rd = -6;
        } else if (MisM7) {
            Rd = -7;
        } else {
            Rd = -8;
        }
    }
```

General exceptions: None

Privilege level: All

3.2. Andes CoDense Extension (XAndesCoDense)

The 16-bit AndeStar CoDense extension is also known as Andes Code Dense Extension.

3.2.1. NDS.EXEC.IT (Execution on Instruction Table)

Format:

15 13	12 9	8	7	6 2	1 0
100	imm[10|4:3|8]	imm[11]	EXEC.IT 0	imm[7:6|2|9|5]	00

Syntax: `NDS.EXEC.IT` *imm[11:2]*

`EXEC.IT` *imm[11:2]* (deprecated)

Purpose: Fetch an instruction from the instruction table and execute the instruction.

Description: This instruction is only available only when `mmsc_cfg.ECD == 1 & mmsc_cfg2.ECDV == 0 & mmsc_cfg.PP16 == 0`. It performs the following two operations:

- Operation 1: It gets a 32-bit instruction data from the instruction table. If the instruction table is hardwired (i.e., `uitb.HW == 1`), *imm[11:2]* is the index of the instruction in the instruction table. Otherwise (i.e., `uitb.HW == 0`), the instruction is in the memory address specified by `[(uitb.ADDR + imm[11:2]) << 2]`.
- Operation 2: If the instruction data is a 32-bit instruction, execute the instruction. Otherwise, an illegal instruction exception is generated.

Operations:

```
If (uitb.HW == 1) {
  Inst = Instruction_Table[imm[11:2]];
  Execute(Inst);
} else {
   VAddr = uitb.ADDR << 2 + Zero_Extend(imm[11:2] << 2);
   (PAddr, Attributes) = Address_Translation(Vaddr);
   Excep_status = Page_Exception(Attributes, POM, Fetch);
   If (Excep_status == NO_EXCEPTION) {
       Inst = Fetch_Memory(PAddr, WORD, Attributes);
       Execute(Inst);
   } else {
       Generate_Exception(Excep_status);
   }
}


Execute (Inst) {
  IF (Inst[1:0] != 3) {
    Generate Illegal Instruction Exception;

  } else {
```

```
        If (Inst.OP == JAL and Rd is x0) {
             PC = concat(PC(XLEN-1,21),imm[20:1]<<1);
         } else if (Inst.OP == JAL and Rd is not x0) {
             Rd = PC + 2;
             PC = concat(PC(XLEN-1,21),imm[20:1]<<1);
         } else if (Inst == JRAL) {
            Rd = PC + 2;
             All the other JRAL operations defined in ISA SPEC;
         } else {
             Execute Inst based on ISA SPEC;
         }
       }
    }
```

Interruptible: Yes

General exceptions: TLB fill, non-leaf PTE not present, leaf PTE not present, read protection, non-execute page, access bit, instruction access fault, bus error, ECC/parity error, illegal instruction

Privilege level: All

Note:

- This instruction is supported only when `mmsc_cfg.ECD` is set. Otherwise, Andes processors generate an illegal instruction exception.
- `Exec.it` is a 16-bit instruction. If the fetched instruction is a 32-bit instruction, its sequential address should be "PC+2". Additionally, if the fetched 32-bit instruction is a non-branch instruction or a not-taken conditional branch, its PC should be updated to "PC+2".
- If the processor encounters an illegal instruction exception while decoding the replaced instruction, the CSR `mtval` is written with the instruction code of the replaced instruction.
- If the processor encounters an instruction access fault exception while fetching the replaced instruction, the CSR `mtval` is written with the address of the replaced instruction that caused the access fault.

3.3. Andes New CoDense Extension (XAndesNewCoDense)

The 16-bit AndeStar New CoDense extension is also known as Andes New Code Dense Extension.

3.3.1. NDS.NEXEC.IT (New Execution on Instruction Table)

Format:

15 13	12	11 9	8	7	6 2	1 0
100	NEXEC.IT 1	imm[4:3|8]	imm[11]	imm[10]	imm[7:6|2|9|5]	00

Syntax: NDS.NEXEC.IT *imm[11:2]*

NEXEC.IT *imm[11:2]* (deprecated)

Purpose: This instruction is an alias of NDS.EXEC.IT with different opcode encoding to resolve opcode conflicts.

Description: This instruction behaves exactly the same as NDS.EXEC.IT. It only exists and replaces NDS.EXEC.IT when mmsc_cfg.ECD == 1 & mmsc_cfg2.ECDV == 1 & mmsc_cfg.PP16 == 0.

3.4. Andes Scalar BFLOAT16 Conversion Extension (XAndesBFHCvt)

This extension defines instructions to perform scalar floating-point conversion between the BFLOAT16 floating-point data and the IEEE-754 32-bit single-precision floating-point (SP) data in a scalar floating-point register.

For RV64, this extension is present if `misa.F == 1 & mmsc_cfg.BF16CVT == 1`; for RV32, it is present if `misa.F == 1 & mmsc_cfg2.BF16CVT == 1`.

3.4.1. NDS.FCVT.S.BF16 (Scalar BF16 to 32-Bit SP Conversion)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
0000000	frs	00010	100	frd	Custom-2 1011011

Syntax: NDS.FCVT.S.BF16 *frd, frs*

FCVT.S.BF16 *frd, frs* (deprecated)

Purpose: Convert BFLOAT16 data to SP data.

Description: This instruction converts BFLOAT16 data in the floating-point register *frs* to SP data and writes the result to the floating-point register *frd*.

The BF16 encoded value is shifted to the left by 16 places and the least significant 16 bits are all written with 0. The result is NaN-boxed by writing the most significant FLEN-32 bits all with 1.

In addition, this instruction will set the invalid operation exception flag (fcsr.NV) if the input operand is a signaling NaN.

Operations:

```
if (frs.H[x] == 0xffff && frs.H[0] != NaN) {
```

```
    frd.H[1] = frs.H[0];
    frd.H[0] = 0;
  } else {
    frd.W[0] = 0x7fc00000;
  }
  frd = NaN-Boxing(frd.W[0]);

  For single-precision floating-point (F extension): x = 1,
  For double-precision floating-point (D extension): x = 3, 2, 1
```

General exceptions: illegal instruction exception

Floating-point exceptions: invalid operation

Privilege level: All

Note:

When multiple floating-point precisions are supported, the valid values of narrower *n*-bit types (where *n* < FLEN) are represented in the lower *n* bits of an FLEN-bit NaN value through a NaN-boxing process. The upper bits of a valid NaN-boxed value must be all 1s.

3.4.2. NDS.FCVT.BF16.S (Scalar 32-Bit SP to BF16 Conversion)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
0000000	frs	00011	100	frd	Custom-2 1011011

Syntax: `NDS.FCVT.BF16.S` *frd, frs*

`FCVT.BF16.S` *frd, frs* (deprecated)

Purpose: Convert SP data to BFLOAT16 data.

Description: This instruction converts SP data in the floating-point register *frs* to BFLOAT16 data and writes the result to the floating-point register *frd*. The rounding mode used by the conversion operation is specified in the `fcsr.frm` field.

In addition, this instruction sets the IEEE-754 exception flags according to IEEE-754 rules.

Operations:

```
frd.H[0] = S_SP_TO_BF16(frs.W[0], fcsr.frm);
frd = Nan-Boxing(frd.H[0]);
```

General exceptions: Illegal instruction exception

Floating-point exceptions: invalid operation, overflow, inexact, underflow

Privilege level: All

3.5. Andes Vector BFLOAT16 Conversion Extension (XAndesVBFHCvt)

This extension defines instructions to perform vector floating-point conversion between the BFLOAT16 floating-point data and the IEEE-754 32-bit single-precision floating-point (SP) data in a vector register.

For RV64, this extension is present if `misa.V == 1 & mvec_cfg.MFSEW != 0 & mmsc_cfg.BF16CVT == 1`; for RV32, it is present if `misa.V == 1 & mvec_cfg.MFSEW != 0 & mmsc_cfg2.BF16CVT == 1`.

3.5.1. NDS.VFWCVT.S.BF16 (Vector BF16 to 32-Bit SP Conversion)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
0000000	vs	00000	100	vd	Custom-2 1011011

Syntax: `NDS.VFWCVT.S.BF16` *vd, vs*

`VFWCVT.S.BF16` *vd, vs* (deprecated)

Purpose: Convert BFLOAT16 data to SP data.

Description: This instruction converts BFLOAT16 data in the vector register *vs* to SP data and writes the result to the vector register *vd*.

This instruction is not masked and it is defined only for SEW=16. If the instruction is executed when SEW is not 16, an illegal instruction exception will be generated.

In addition, this instruction will set the invalid operation exception flag (`fcsr.NV`) if the input operand is a signaling NaN.

Operations:

```
if (vs[i] != NAN) {
```

```
    vd[i].H[1] = vs[i];
    vd[i].H[0] = 0;
  } else {
    vd[i] = 0x7fc00000;
  }
```

General exceptions: illegal instruction exception

Floating-point exceptions: invalid operation

Privilege level: All

3.5.2. NDS.VFNCVT.BF16.S (Vector 32-Bit SP to BF16 Conversion)

Format:

31 25	24 20	19 15	14 12	11 7	6 0
0000000	vs	00001	100	vd	Custom-2 1011011

Syntax: `NDS.VFNCVT.BF16.S` *vd, vs*

`VFNCVT.BF16.S` *vd, vs* (deprecated)

Purpose: Convert SP data to BFLOAT16 data.

Description: This instruction converts SP data in the vector register *vs* to BFLOAT16 data and writes the result to the vector register *vd*. The rounding mode used by the conversion operation is specified in the `fcsr.frm` field.

This instruction is not masked and it is defined only for SEW=16. If the instruction is executed when SEW is not 16, an illegal instruction exception will be generated.

In addition, this instruction sets the IEEE-754 exception flags according to IEEE-754 rules.

Operations:

```
vd[i] = V_SP_TO_BF16(vs[i], fcsr.frm);
```

General exceptions: illegal instruction exception

Floating-point exceptions: invalid operation, overflow, inexact, underflow

Privilege level: All

3.6. Andes Vector INT4 Load Extension (XAndesVSIntLoad)

This extension defines vector load instructions to move sign-extended or zero-extended INT4 data into 8-bit vector register elements.

For RV64, this extension is present if `misa.V == 1 & mmsc_cfg.VL4 == 1`; for RV32, it is present if `misa.V == 1 & mmsc_cfg2.VL4 == 1`.

3.6.1. NDS.VLN8.V (Vector Signed 4-Bit Uni-Stride Load into 8-Bit Element)

Format:

31 26	25	24 20	19 15	14 12	11 7	6 0
000001	vm	00010	rs1	100	vd	Custom-2 1011011

Syntax: NDS.VLN8.V *vd, (rs1), vm*

VLN8.V *vd, (rs1), vm* (deprecated)

Purpose: Load a vector length (VL) number of sign-extended INT4 data from the memory into a vector register of 8-bit elements with a uni-stride address.

Description: This instruction loads a continuous VL number of signed-extended INT4 data from the memory into the vector register *vd* in which each element is 8-bit wide. The starting address for the memory access is specified in the register *rs1* and the instruction can be masked using the register *vm*.

Operations:

```
Element_size = 8;
i = VSTART..(VL-1)
addr = rs1 + floor(i/2);
part = i%2;
nibble[3:0] = MEM(addr)[4*part+3:4*part+0];
```

```
if (vm.E[i] == 1) {
    vd.E[i] = Sign-Extend(nibble[3:0]);
}
```

General exceptions: load access fault, load page fault

Privilege level: All

3.6.2. NDS.VLNU8.V (Vector Unsigned 4-Bit Uni-Stride Load into 8-Bit Element)

Format:

31 26	25	24 20	19 15	14 12	11 7	6 0
000001	vm	00011	rs1	100	vd	Custom-2 1011011

Syntax: NDS.VLNU8.V *vd, (rs1), vm*

VLNU8.V *vd, (rs1), vm* (deprecated)

Purpose: Load a VL number of zero-extended INT4 data from the memory into a vector register of 8-bit elements with a uni-stride address.

Description: This instruction loads a continuous VL number of zero-extended INT4 data from the memory into the vector register *vd* in which each element is 8-bit wide. The starting address for the memory access is specified in the register *rs1* and the instruction can be masked using the register *vm*.

Operations:

```
Element_size = 8;
i = VSTART..(VL-1)
addr = rs1 + floor(i/2);
part = i%2;
nibble[3:0] = MEM(addr)[4*part+3:4*part+0];
```

```
if (vm.E[i] == 1) {
    vd.E[i] = Zero-Extend(nibble[3:0]);
}
```

General exceptions: load access fault, load page fault

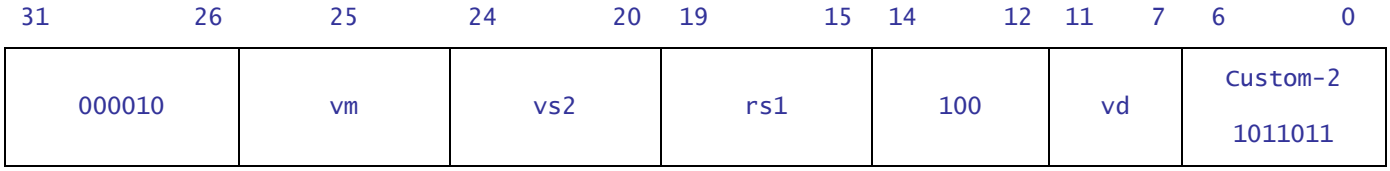
Privilege level: All

3.7. Andes Vector Packed FP16 Extension (XAndesVPackFPH)

This extension is present if `mmsc_cfg.VPFH == 1`.

3.7.1. NDS.VFPMADT.VF (Vector Single-Width Floating-Point Packed Fused Multiply-Add with Top FP16 as Multiplicand)

Format:



Syntax: `NDS.VFPMADT.VF` *vd, rs1, vs2, vm*

`VFPMADT.VF` *vd, rs1, vs2, vm* (deprecated)

Purpose: Extract a pair of FP16 data from a floating-point register. Multiply the top FP16 data with the FP16 elements and add the result with the bottom FP16 data.

Description: This instruction extracts a pair of FP16 data from the source scalar floating-point register *rs1* and multiplies the top FP16 data in the pair *rs1.FP16[1]* with the FP16 elements in the vector register *vs2*. The multiplication results are added with the bottom FP16 data in the pair *rs1.FP16[0]* next and the element addition results are written back to the register *vd*.

This instruction is defined only for SEW=16. If the instruction is executed when SEW is not 16, an illegal instruction exception will be generated.

Operations:

```
vd[i] = (f[rs1].FP16[1] * vs2[i]) + f[rs1].FP16[0];
```

General exceptions: illegal instruction exception

Floating-point exceptions: N/A

Privilege level: All

3.7.2. NDS.VFPMADB.VF (Vector Single-Width Floating-Point Packed Fused Multiply-Add with Bottom FP16 as Multiplicand)

Format:

31 26	25	24 20	19 15	14 12	11 7	6 0
000011	vm	vs2	rs1	100	vd	Custom-2 1011011

Syntax: NDS.VFPMADB.VF *vd, rs1, vs2, vm*

VFPMADB.VF *vd, rs1, vs2, vm* (deprecated)

Purpose: Extract a pair of FP16 data from a floating-point register. Multiply the bottom FP16 data with the FP16 elements and add the result with the top FP16 data.

Description: This instruction extracts a pair of FP16 data from the source scalar floating-point register *rs1* and multiplies the bottom FP16 data in the pair *rs1.FP16[0]* with the FP16 elements in the vector register *vs2*. The multiplication results are next added with the top FP16 data in the pair *rs1.FP16[1]* and the element addition results are written back to the register *vd*.

This instruction is defined only for SEW=16. If the instruction is executed when SEW is not 16, an illegal instruction exception will be generated.

Operations:

```
vd[i] = (f[rs1].FP16[0] * vs2[i]) + f[rs1].FP16[1];
```

General exceptions: illegal instruction exception

Floating-point exceptions: N/A

Privilege level: All

3.8. Andes Vector Dot Product Extension (XAndesVDot)

For RV64, this extension is present if `misa.V == 1 & mmsc_cfg.VDOT == 1`; for RV32, it is present if `misa.V == 1 & mmsc_cfg2.VDOT == 1`.

3.8.1. NDS.VD4DOTS.VV (Vector Signed Dot Product on 1/4 of SEW)

Format:

31 26	25	24 20	19 15	14 12	11 7	6 0
000100	vm	vs2	vs1	100	vd	Custom-2 1011011

Syntax: NDS.VD4DOTS.VV *vd, vs1, vs2, vm*

VD4DOTS.VV *vd, vs1, vs2, vm* (deprecated)

Purpose: Calculate the signed dot product of four SEW/4-bit data and accumulate the result into a SEW-bit element for all elements in a vector register.

Description: This instruction calculates the signed dot product of four sets of SEW/4-bit data between the elements of the register *vs1* (signed) and those of another register *vs2* (signed) and produces signed SEW-bit results. The results are accumulated into the corresponding elements of the destination vector register *vd*.

This instruction is defined only for SEW=32 or 64. If the instruction is executed when SEW is not 16 or 64, an illegal instruction exception will be generated.

Operations:

```
D4 = SEW/4;

vs1d0[i] = vs1[i].D4[0]; vs2d0[i] = vs2[i].D4[0];
```

```
vs1d1[i] = vs1[i].D4[1]; vs2d1[i] = vs2[i].D4[1];

vs1d2[i] = vs1[i].D4[2]; vs2d2[i] = vs2[i].D4[2];

vs1d3[i] = vs1[i].D4[3]; vs2d3[i] = vs2[i].D4[3];

vd[i] = vd[i] + vs1d0[i] s* vs2d0[i] + vs1d1[i] s* vs2d1[i] +

                vs1d2[i] s* vs2d2[i] + vs1d3[i] s* vs2d3[i];
```

General exceptions: illegal instruction exception

Privilege level: All

3.8.2. NDS.VD4DOTU.VV (Vector Unsigned Dot Product on 1/4 of SEW)

Format:

31 26	25	24 20	19 15	14 12	11 7	6 0
000111	vm	vs2	vs1	100	vd	Custom-2 1011011

Syntax: NDS.VD4DOTU.VV *vd, vs1, vs2, vm*

VD4DOTU.VV *vd, vs1, vs2, vm* (deprecated)

Purpose: Calculate the unsigned dot product of four SEW/4-bit data and accumulate the result into a SEW-bit element for all elements in a vector register.

Description: This instruction calculates the unsigned dot product of four sets of SEW/4-bit data between the elements of the register *vs1* (unsigned) and those of another register *vs2* (unsigned) and produces unsigned SEW-bit results. The results are accumulated into the corresponding elements of the destination vector register *vd*.

This instruction is only defined for SEW=32 or 64. If the instruction is executed when SEW is not 32 or 64, an illegal instruction exception will be generated.

Operations:

```
D4 = SEW/4;
vs1d0[i] = vs1[i].D4[0]; vs2d0[i] = vs2[i].D4[0];
```

```
vs1d1[i] = vs1[i].D4[1]; vs2d1[i] = vs2[i].D4[1];

vs1d2[i] = vs1[i].D4[2]; vs2d2[i] = vs2[i].D4[2];

vs1d3[i] = vs1[i].D4[3]; vs2d3[i] = vs2[i].D4[3];

vd[i] = vd[i] + vs1d0[i] u* vs2d0[i] + vs1d1[i] u* vs2d1[i] +

                vs1d2[i] u* vs2d2[i] + vs1d3[i] u* vs2d3[i];
```

General exceptions: illegal instruction exception

Privilege level: All

3.8.3. NDS.VD4DOTSU.VV (Vector Signed and Unsigned Dot Product on 1/4 of SEW)

Format:

31 26	25	24 20	19 15	14 12	11 7	6 0
000101	vm	vs2	vs1	100	vd	Custom-2 1011011

Syntax: NDS.VD4DOTSU.VV *vd, vs1, vs2, vm*

VD4DOTSU.VV *vd, vs1, vs2, vm* (deprecated)

Purpose: Calculate the signed and unsigned dot product of four SEW/4-bit data and accumulate the result into a SEW-bit element for all elements in a vector register.

Description: This instruction calculates the signed and unsigned dot product of four sets of SEW/4-bit data between the elements of the register *vs1* (signed) and those of another register *vs2* (unsigned) and produces signed SEW-bit results. The results are accumulated into the corresponding elements of the destination vector register *vd*.

This instruction is defined only for SEW=32 or 64. If this instruction is executed when SEW is not 16 or 64, an illegal instruction exception will be generated.

Operations:

```
D4 = SEW/4;

vs1d0[i] = vs1[i].D4[0]; vs2d0[i] = vs2[i].D4[0];
```

```
vs1d1[i] = vs1[i].D4[1]; vs2d1[i] = vs2[i].D4[1];

vs1d2[i] = vs1[i].D4[2]; vs2d2[i] = vs2[i].D4[2];

vs1d3[i] = vs1[i].D4[3]; vs2d3[i] = vs2[i].D4[3];

vd[i] = vd[i] + vs1d0[i] su* vs2d0[i] + vs1d1[i] su* vs2d1[i] +

                vs1d2[i] su* vs2d2[i] + vs1d3[i] su* vs2d3[i];
```

General exceptions: illegal instruction exception

Privilege level: All

3.9. Andes Vector Small INT Handling Extension (XAndesVSIntH)

For RV64, this extension is present if `misa.V == 1 & mmsc_cfg.VSIH == 1`; for RV32, it is present if `misa.V == 1 & mmsc_cfg2.VSIH == 1`.

The behavior of the following vector integer extension instructions will be extended by this extension.

The source operand EEW of these instructions will be extended to 4.

vzext.vf2
vsext.vf2
vzext.vf4
vsext.vf4
vzext.vf8
vsext.vf8

3.9.1. NDS.VLE4.V (Vector 4-Bit Uni-Stride Load into a Vector Register)

Format:

31 26	25	24 20	19 15	14 12	11 7	6 0
000001	1	00000	rs1	100	vd	Custom-2 1011011

Syntax: `NDS.VLE4.V` *vd, (rs1)*

`VLE4.V` *vd, (rs1)* (deprecated)

Purpose: Load a VL number of INT4 data from the memory into a vector register with a uni-stride address.

Description: This instruction loads a continuous vector length number of INT4 data from the memory into the vector register `vd`. The effective element width (EEW) of this instruction is 4 and the effective LMUL (EMUL) is (4/SEW)*LMUL. The starting address for the memory access is specified in the register *rs1* and the instruction is not masked.

In addition, when the vector length is an odd number and `vtype.vta == 0` (i.e., vector tail undisturbed), this instruction will not conform to the undisturbed behavior for certain tail elements.

Operations:

```
Element_size = 4;
```

```
VSTART = VSTART - (VSTART % 2);

i = VSTART..(VL-1)

addr = rs1 + floor(i/2);

part = i%2;

nibble[3:0] = MEM(addr)[4*part+3:4*part+0];

vd.E[i] = nibble[3:0];
```

General exceptions: Load access fault, Load page fault

Privilege level: All

3.9.2. NDS.VFWCVT.F.N.V (Vector Signed INT4 to SEW FP Conversion)

Format:

31 26	25	24 20	19 15	14 12	11 7	6 0
000000	vm	vs	00100	100	vd	Custom-2 1011011

Syntax: NDS.VFWCVT.F.N.V *vd, vs, vm*

VFWCVT.F.N.V *vd, vs, vm* (deprecated)

Purpose: Convert signed INT4 data to SEW-sized floating-point data.

Description: This instruction converts signed INT4 data in the source vector register *vs* to SEW-sized floating-point data and writes the result to the destination vector register *vd*. The source EEW is 4 and the source EMUL is (4/SEW)*LMUL. The destination has its EEW equal to SEW and EMUL equal to LMUL.

This instruction is defined only for SEW=16 or 32. If the instruction is executed when SEW is not 16 or 32, an illegal instruction exception will be generated.

The following table lists the constraints on the source vector register group (*vs*) based on the LMUL of the destination register group (*vd*).

LMUL (*vd*)	EMUL (*vs*)	
	SEW == 16	SEW == 32
8	2	1
4	1	1/2
2	1/2	1/4
1	1/4	1/8
1/2	1/8	Reserved
1/4	Reserved	Reserved
1/8	Reserved	Reserved

Operations:

```
vd[i] = Convert_SINT4_to_FP(vs[i], SEW);  // vfwcvt.f.n.v
```

General exceptions: illegal instruction exception

Floating-point exceptions: N/A

Privilege level: All

3.9.3. NDS.VFWCVT.F.NU.V (Vector Unsigned INT4 to SEW FP Conversion)

Format:

31 26	25	24 20	19 15	14 12	11 7	6 0
000000	vm	vs	00101	100	vd	Custom-2 1011011

Syntax: NDS.VFWCVT.F.NU.V *vd, vs, vm*

VFWCVT.F.NU.V *vd, vs, vm* (deprecated)

Purpose: Convert unsigned INT4 data to SEW-sized floating-point data.

Description: This instruction converts unsigned INT4 data in the vector register *vs* to SEW-sized floating-point data and writes the result to the vector register *vd*. The source EEW is 4 and the source EMUL is (4/SEW)*LMUL. The destination has its EEW equal to SEW and EMUL equal to LMUL.

This instruction is defined only for SEW=16 or 32. If the instruction is executed when SEW is not 16 or 32, an illegal instruction exception will be generated.

The following table lists the constraints on the source vector register group (*vs*) based on the LMUL of the destination register group (*vd*).

LMUL (*vd*)	EMUL (*vs*)	
	SEW == 16	SEW == 32
8	2	1
4	1	1/2
2	1/2	1/4
1	1/4	1/8
1/2	1/8	Reserved
1/4	Reserved	Reserved
1/8	Reserved	Reserved

Operations:

```
vd[i] = Convert_UINT4_to_FP(vs[i], SEW);  // vfwcvt.f.nu.v
```

General exceptions: illegal instruction exception

Floating-point exceptions: N/A

Privilege level: All

3.9.4. NDS.VFWCVT.F.B.V (Vector Signed INT8 to SEW FP Conversion)

Format:

31 26	25	24 20	19 15	14 12	11 7	6 0
000000	vm	vs	00110	100	vd	Custom-2 1011011

Syntax: NDS.VFWCVT.F.B.V *vd, vs, vm*

VFWCVT.F.B.V *vd, vs, vm* (deprecated)

Purpose: Convert signed INT8 data to SEW-sized floating-point data.

Description: This instruction converts signed INT8 data in the vector register *vs* to SEW-sized floating-point data and writes the result to the vector register *vd*. The source EEW is 8 and the source EMUL is (8/SEW)*LMUL. The destination has its EEW equal to SEW and EMUL equal to LMUL.

This instruction is defined only for SEW=16 or 32. If the instruction is executed when SEW is not 16 or 32, an illegal instruction exception will be generated.

The following table lists the constraints on the source vector register group (*vs*) based on the LMUL of the destination register group (*vd*).

LMUL (*vd*)	EMUL (*vs*)	
	SEW == 16	SEW == 32
8	4	2
4	2	1
2	1	1/2
1	1/2	1/4
1/2	1/4	1/8
1/4	1/8	Reserved
1/8	Reserved	Reserved

Operations:

```
vd[i] = Convert_UINT8_to_FP(vs[i], SEW);  // vfwcvt.f.bu.v
```

General exceptions: illegal instruction exception

Floating-point exceptions: N/A

Privilege level: All

3.9.5. NDS.VFWCVT.F.BU.V (Vector Unsigned INT8 to SEW FP Conversion)

Format:

31 26	25	24 20	19 15	14 12	11 7	6 0
000000	vm	vs	00111	100	vd	Custom-2 1011011

Syntax: NDS.VFWCVT.F.BU.V *vd, vs, vm*

VFWCVT.F.BU.V *vd, vs, vm* (deprecated)

Purpose: Convert unsigned INT8 data to SEW-sized floating-point data.

Description: This instruction converts unsigned INT8 data in the vector register *vs* to SEW-sized floating-point data and writes the result to the vector register *vd*. The source EEW is 8 and the source EMUL is (8/SEW)*LMUL. The destination has its EEW equal to SEW and EMUL equal to LMUL.

This instruction is defined only for SEW=16 or 32. If the instruction is executed when SEW is not 16 or 32, an illegal instruction exception will be generated.

The following table lists the constraints on the source vector register group (*vs*) based on the LMUL of the destination register group (*vd*).

LMUL (*vd*)	EMUL (*vs*)	
	SEW == 16	SEW == 32
8	4	2
4	2	1
2	1	1/2
1	1/2	1/4
1/2	1/4	1/8
1/4	1/8	Reserved
1/8	Reserved	Reserved

Operations:

```
vd[i] = Convert_UINT8_to_FP(vs[i], SEW);  // vfwcvt.f.bu.v
```

General exceptions: illegal instruction exception

Floating-point exceptions: N/A

Privilege level: All

3.10. Andes Vector Quad-Widening Integer Multiply-Add Extension (XAndesVQMac)

The quad-widening integer multiply-add instructions add a SEW-bit*SEW-bit multiply result to/from a 4*SEW-bit value and produce a 4*SEW-bit result. They support all combinations of signed and unsigned multiply operands.

For RV64, this extension is present if `misa.V == 1 & mrvarch_cfg.Zvqmac == 1`; for RV32, it is present if `misa.V == 1 & mrvarch_cfg2.Zvqmac == 1`.

3.10.1. NDS.VQMACCU.VV (Quad-Widening Unsigned-Integer Multiply-Add, Overwrite Addend)

Format:

31 26	25	24 20	19 15	14 12	11 7	6 0
111100	vm	vs2	vs1	000	vd	1010111

Syntax: NDS.VQMACCU.VV *vd, vs1, vs2, vm*

VQMACCU.VV *vd, vs1, vs2, vm* (deprecated)

Purpose: Multiply two SEW-bit values and accumulate the produced 4*SEW-bit result.

Description: This instruction multiplies an unsigned SEW-bit value in the vector register *vs1* by an unsigned SEW-bit value in the vector register *vs2* to produce an unsigned 4*SEW-bit result. In addition, it accumulates the produced 4*SEW-bit result to the vector register *vd*.

On ELEN=32 machines, only 8b * 8b = 16b products accumulated in a 32b accumulator are supported. Machines with ELEN=64 also support 16b * 16b = 32b products accumulated in a 64b accumulator. If SEW is not a legal value when this instruction is executed, an illegal instruction exception will be generated.

Operations:

```
vd[i] = +(vs1[i] * vs2[i]) + vd[i];
```

General exceptions: illegal instruction exception

Privilege level: All

3.10.2. NDS.VQMACCU.VX (Quad-Widening Unsigned-integer Multiply-Add, Overwrite Addend)

Format:

31 26	25	24 20	19 15	14 12	11 7	6 0
111100	vm	vs2	rs1	100	vd	1010111

Syntax: NDS.VQMACCU.VX *vd, rs1, vs2, vm*

VQMACCU.VX *vd, rs1, vs2, vm* (deprecated)

Purpose: Multiply two SEW-bit values and accumulate the produced 4*SEW-bit result.

Description: This instruction multiplies an unsigned SEW-bit value in the scalar register *rs1* by an unsigned SEW-bit value in the vector register *vs2* to produce an unsigned 4*SEW-bit result. In addition, it accumulates the produced 4*SEW-bit result to the vector register vd.

On ELEN=32 machines, only 8b * 8b = 16b products accumulated in a 32b accumulator are supported. Machines with ELEN=64 also support 16b * 16b = 32b products accumulated in a 64b accumulator. If SEW is not a legal value when this instruction is executed, an illegal instruction exception will be generated.

Operations:

```
vd[i] = +(x[rs1] * vs2[i]) + vd[i];
```

General exceptions: illegal instruction exception

Privilege level: All

3.10.3. NDS.VQMACC.VV (Quad-Widening Signed-Integer Multiply-Add, Overwrite Addend)

Format:

31 26	25	24 20	19 15	14 12	11 7	6 0
111101	vm	vs2	vs1	000	vd	1010111

Syntax: NDS.VQMACC.VV *vd, vs1, vs2, vm*

VQMACC.VV *vd, vs1, vs2, vm* (deprecated)

Purpose: Multiply two SEW-bit values and accumulate the produced 4*SEW-bit result.

Description: This instruction multiplies a signed SEW-bit value in the vector register *vs1* by a signed SEW-bit value in the vector register *vs2* to produce a signed 4*SEW-bit result. In addition, it accumulates the produced 4*SEW-bit result to the vector register *vd*.

On ELEN=32 machines, only 8b * 8b = 16b products accumulated in a 32b accumulator are supported. Machines with ELEN=64 also support 16b * 16b = 32b products accumulated in a 64b accumulator. If SEW is not a legal value when this instruction is executed, an illegal instruction exception will be generated.

Operations:

```
vd[i] = +(vs1[i] * vs2[i]) + vd[i];
```

General exceptions: illegal instruction exception

Privilege level: All

3.10.4. NDS.VQMACC.VX (Quad-Widening Signed-Integer Multiply-Add, Overwrite Addend)

Format:

31 26	25	24 20	19 15	14 12	11 7	6 0
111101	vm	vs2	rs1	100	vd	1010111

Syntax: NDS.VQMACC.VX *vd, rs1, vs2, vm*

VQMACC.VX *vd, rs1, vs2, vm* (deprecated)

Purpose: Multiply two SEW-bit values and accumulate the produced 4*SEW-bit result.

Description: This instruction multiplies a signed SEW-bit value in the scalar register *rs1* by a signed SEW-bit value in the vector register *vs2* to produce a signed 4*SEW-bit result. In addition, it accumulates the produced 4*SEW-bit result to the vector register *vd*.

On ELEN=32 machines, only 8b * 8b = 16b products accumulated in a 32b accumulator are supported. Machines with ELEN=64 also support 16b * 16b = 32b products accumulated in a 64b accumulator. If SEW is not a legal value when this instruction is executed, an illegal instruction exception will be generated.

Operations:

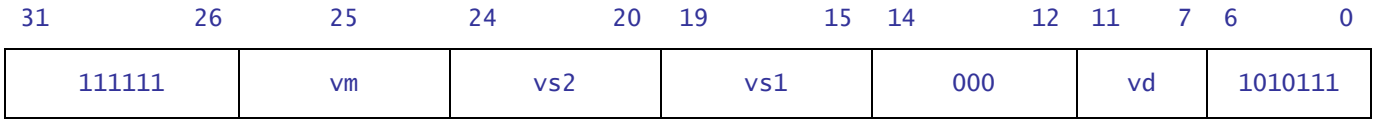
```
vd[i] = +(x[rs1] * vs2[i]) + vd[i];
```

General exceptions: illegal instruction exception

Privilege level: All

3.10.5. NDS.VQMACCSU.VV (Quad-Widening Signed-Unsigned-Integer Multiply-Add, Overwrite Addend)

Format:



Syntax: NDS.VQMACCSU.VV *vd, vs1, vs2, vm*

VQMACCSU.VV *vd, vs1, vs2, vm* (deprecated)

Purpose: Multiply two SEW-bit values and accumulate the produced 4*SEW-bit result.

Description: This instruction multiplies a signed SEW-bit value in the vector register *vs1* by an unsigned SEW-bit value in the vector register *vs2* to produce a signed 4*SEW-bit result. In addition, it accumulates the produced 4*SEW-bit result to the vector register *vd*.

On ELEN=32 machines, only 8b * 8b = 16b products accumulated in a 32b accumulator are supported. Machines with ELEN=64 also support 16b * 16b = 32b products accumulated in a 64b accumulator. If SEW is not a legal value when this instruction is executed, an illegal instruction exception will be generated.

Operations:

```
vd[i] = +(signed(vs1[i]) * unsigned(vs2[i])) + vd[i];
```

General exceptions: illegal instruction exception

Privilege level: All

3.10.6. NDS.VQMACCSU.VX (Quad-Widening Signed-Unsigned-Integer Multiply-Add, Overwrite Addend)

Format:

31 26	25	24 20	19 15	14 12	11 7	6 0
111111	vm	vs2	rs1	100	vd	1010111

Syntax: NDS.VQMACCSU.VX *vd, rs1, vs2, vm*

VQMACCSU.VX *vd, rs1, vs2, vm* (deprecated)

Purpose: Multiply two SEW-bit values and accumulate the produced 4*SEW-bit result.

Description: This instruction multiplies a signed SEW-bit value in the scalar register *rs1* by an unsigned SEW-bit value in the vector register *vs2* to produce a signed 4*SEW-bit result. In addition, it accumulates the produced 4*SEW-bit result to the vector register *vd*.

On ELEN=32 machines, only 8b * 8b = 16b products accumulated in a 32b accumulator are supported. Machines with ELEN=64 also support 16b * 16b = 32b products accumulated in a 64b accumulator. If SEW is not a legal value when this instruction is executed, an illegal instruction exception will be generated.

Operations:

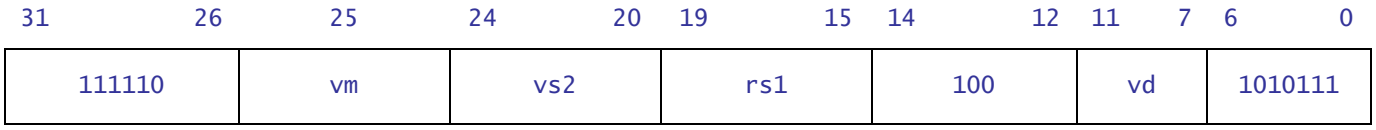
```
vd[i] = +(signed(x[rs1]) * unsigned(vs2[i])) + vd[i];
```

General exceptions: illegal instruction exception

Privilege level: All

3.10.7. NDS.VQMACCUS.VX (Quad-Widening Unsigned-Signed-Integer Multiply-Add, Overwrite Addend)

Format:



31 26	25	24 20	19 15	14 12	11 7	6 0
111110	vm	vs2	rs1	100	vd	1010111

Syntax: NDS.VQMACCUS.VX *vd, rs1, vs2, vm*

VQMACCUS.VX *vd, rs1, vs2, vm (*deprecated)

Purpose: Multiply two SEW-bit values and accumulate the produced 4*SEW-bit result.

Description: This instruction multiplies an unsigned SEW-bit value in the scalar register *rs1* by a signed SEW-bit value in the vector register *vs2* to produce a signed 4*SEW-bit result. In addition, it accumulates the produced 4*SEW-bit result to the vector register *vd*.

On ELEN=32 machines, only 8b * 8b = 16b products accumulated in a 32b accumulator are supported. Machines with ELEN=64 also support 16b * 16b = 32b products accumulated in a 64b accumulator. If SEW is not a legal value when this instruction is executed, an illegal instruction exception will be generated.

Operations:

```
vd[i] = +(unsigned(x[rs1]) * signed(vs2[i])) + vd[i];
```

General exceptions: illegal instruction exception

Privilege level: All

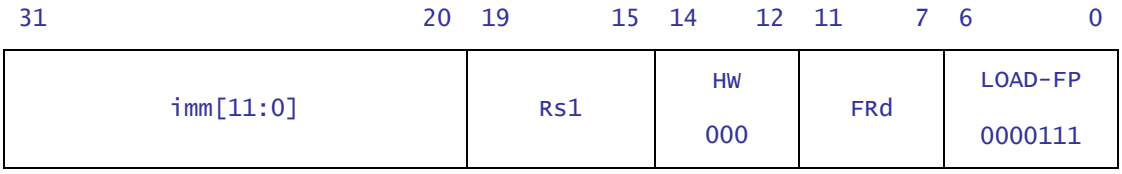
Appendix: Obsolete Extensions and Instructions

This section provides information about the obsolete extensions and their associated instructions.

Appendix I. Andes Half-Precision Floating-Point Extension

Appendix I-I. FLHW (Floating-point Load from Half-Precision to Single-Precision)

Format:



Syntax: FLHW *FRd, imm[11:0] (Rs1)*

Purpose: Load half-precision (16-bit) floating-point data from the memory and convert it to single-precision data.

Description: This instruction loads half-precision (16-bit) floating-point data from the memory, converts it to single-precision format, and then writes the result into the floating-point register *FRd*. The memory address is specified by a base address in *Rs1* plus a 12-bit signed byte offset, *imm[11:0]*.

If the loaded half-precision floating-point data is a signaling NaN (i.e., [14:10]=0x1F, [9]=0, [8:0] ≠0), this instruction will raise the invalid operation status flag (i.e., fcsr.NV) and produce

a single-precision canonical NaN (i.e., 0x7FC00000).

Operations:

```
Vaddr = Rs1 + Sign_Extend(imm[11:0]);
(PAddr, Attributes) = Address_Translation(Vaddr);
Excep_status = Page_Exception(Attributes, POM, LOAD);
If (Excep_status == NO_EXCEPTION) {
   Hdata[15:0] = Load_Memory(PAddr, HWORD, Attributes);
   FRd[31:0] = Convert_HP_to_SP(Hdata[15:0]);
} else {
   Generate_Exception(Excep_status);
}
```

General exceptions: load address misaligned, load access fault, load page fault, bus error, ECC/parity error

Floating-point exceptions: invalid operation

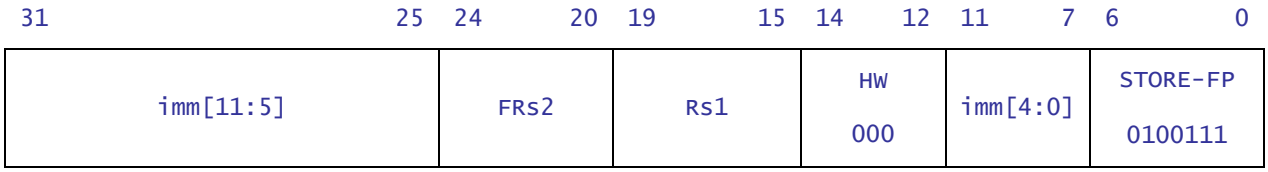
Privilege level: All

Note:

- Under the RISC-V ISA specification, the underflow checking is performed by detecting tininess after rounding the operation.

Appendix I-II. FSHW (Floating-point Store to Half-Precision from Single-Precision)

Format:



Syntax: FSHW *FRs2, imm[11:0] (Rs1)*

Purpose: Store half-precision (16-bit) floating-point data converted from single-precision data to the memory.

Description: This instruction converts single-precision data in the register *FRs2* to half-precision floating-point data and stores the result to the memory. The address for the memory access is specified by a base address in Rs1 plus a 12-bit signed byte offset, *imm[11:0]*.

The behaviors of this instruction for converting a single-precision value to a half-precision value are defined as follows.

- The rounding mode is “Round towards Zero”.
- If an overflow condition occurs, the rounded value is saturated to the value of $\pm 2^{15} \cdot (1+2^{-10} \cdot (0x3FF))$. The overflow exception flag will then be set in the fcsr.OF bit and the inexact exception flag in the fcsr.NX bit.
- If an underflow conditio occurs, the rounded value, *v*, might be delivered as a subnormal value if it is within the subnormal range (i.e., $1.0x2^{-14} > v \geqq 1.0x2^{-24}$), or a signed zero if $v < 1.0x2^{-24}$. If

the delivered value is inexact, the underflow exception flag will be set in the fcsr.UF bit and the inexact exception flag in the fcsr.NX bit.

- If the single-precision value is a signaling NaN (i.e., [30:23]=0xFF, [22]=1, [21:0]≠0), this instruction will raise the invalid operation status flag (i.e., fcsr.NV) and produce a half-precision canonical NaN (i.e., 0x7E00).

Operations:

```
Vaddr = Rs1 + Sign_Extend(imm[11:0]);
(PAddr, Attributes) = Address_Translation(Vaddr);
Excep_status = Page_Exception(Attributes, POM, LOAD);
If (Excep_status == NO_EXCEPTION) {
   Hdata[15:0] = Convert_SP_to_HP(FRs2[31:0])
   Store_Memory(PAddr, HWORD, Attributes, Hdata[15:0]);
} else {
   Generate_Exception(Excep_status);
}
```

General exceptions: store address misaligned, store access fault, store page fault, bus error, ECC/parity error

Floating-point exceptions: inexact, overflow, underflow, invalid operation

Privilege level: All

Note:

- Under the RISC-V ISA specification, the underflow checking is performed by detecting the tininess after rounding the operation.
- The IEEE 754-2008 standard does not regard the detection of an exact subnormal as an underflow condition.